4-26-1999

Designing an Integrated Environment for Artificial Intelligence

Andrew B. Ritger '99
Illinois Wesleyan University

Designing an Integrated Environment for Artificial Intelligence

Andrew B. Ritger and Dr. Lionel R. Shapiro
Department of Computer Science and Mathematics
Illinois Wesleyan University

April 26, 1999

Contents

List of Figures

Abstract

The SHELLEY RESEARCH GROUP (part of the Illinois Wesleyan Intelligence Network on Knowledge - IWINK) has been in existence for several years, and has benefited immensely from various student contributors who have added such components as robotic arm control, cross platform networking, an artificially intelligent tic-tac-toe player, and an interactive teaching tool demonstrating the functionality of artificial neural networks. What is lacking, however, amidst these undergraduate contributions to the SHELLEY Project, is an effective means of integrating existing components into a single cohesive functional unit, let alone any easy means of making further contributions within a simple unified context.

The focus of this research has been to design an all-encompassing structure for incorporating the different components of SHELLEY (both existing and future). Because we must operate under the assumption that we cannot predict what future contributions will be made to SHELLEY, nor how these components will be used, this integrated environment must be both flexible and expandable in such a way as to not confine future projects.

The approach to artificial intelligence that the SHELLEY RESEARCH GROUP has taken relies heavily upon interaction with the surrounding environment. For this reason, many of the existing components are devices for receiving input from SHELLEY's surroundings (such as vision cameras) or acting upon the surroundings (such as robotic arms). Thus, we can assume that future contributions will fall under two primary categories: additional devices (either cognitive modules, such as neural networks, or interactive devices, such as cameras or arms), or intelligent agents (such as tic-tac-toe players, or navigation systems) that will use these devices. The environment must then be flexible in two manners – allowing for the addition of further devices, and providing a task management mechanism for accessing these devices. The solution is to use a modern operating system model where the devices that SHELLEY uses to interact with her environment correspond to computer hardware devices and their drivers, the intelligent agents are analogous to processes that run on the system and use the devices, and the administrator, which coordinates these agents and their usage of devices, can be compared to the kernel of the modern operating system.

1 Motivations

The primary purpose of this research has been to ease the implementation process for future students making contributions to the SHELLEY Project.† It is currently difficult for students to make contributions because each external robotic device has its own unique communication protocol. Additionally, even in the case where there exists a program that communicates with an external device (for example, a robotic arm), there is no system in place to facilitate other programs reusing that same portion of program code. Finally, a third obstacle hindering progress is intimidation felt by potential contributors. Students often examine the prospects of implementing a task for SHELLEY such as playing tic-tac-toe or chess, but may feel that the project is too daunting because "that's so complicated" or "I don't know anything about robotics." By providing a simple interface to an all-encompassing structure for incorporating the different components of SHELLEY, these difficulties can be alleviated.

2 The Requirements of an Integrated Environment for Artificial Intelligence

An *agent* is an entity that perceives characteristics of its environment and acts upon that environment. An *intelligent* agent acts upon its environment in ways humans consider appropriate to the characteristics that the agent perceives. It uses its *sensors* – means of perceiving its surroundings – to collect information that it then uses to make intelligent decisions. The agent then acts upon its surroundings through its *effectors* [1]. It is in this decision making, or *mapping* of input from sensors to output actions through the effectors, that the intelligence of the agent lies. This mechanism for agent intelligence, however, is not the focus of this research; there are many different approaches and techniques for making an agent intelligent, which encompasses several major paradigms and philosophies. It is also not the place of this research to make a judgment as to which paradigm is most appropriate for SHELLEY, but rather to design an environment that can facilitate different approaches to building an intelligent agent so that future students can explore the many options without feeling confined or restrained to one predefined paradigm.

The distinction can be made between pure software agents, whose world consists entirely of entities internal to the computer upon which the agent resides (*soft agents*), and agents whose world extends beyond the confines of a computer and encompasses the physical parameters of its surroundings [1]. In the former, the sensors and effectors are much more easily implemented, while the latter requires special hardware that introduces all the complications already discussed (see **Section 1 Motivations**).

SHELLEY is a robotic entity – her environment is the physical world. Therefore, while some tasks may be handled sufficiently through soft agents, others must be addressed by agents who make use of SHELLEY's special hardware peripherals in order to interact with the "real world" [2].

†The SHELLEY Project takes its name from *Frankenstein* author Mary Shelley

In light of the aforementioned difficulties associated with interfacing to these peripherals, we require some mechanism through which agents can easily access specialized external hardware in order to accomplish their tasks. Thus, SHELLEY necessitates an *integrated environment* that can provide an effective and flexible system for integrating both existing and future peripherals such that these devices can be shared and adequately managed. Additionally, this integrated environment must provide a simple method of programming with and using these peripherals. Ideally, this can exist in the form of function calls which can be made directly from within researchers' program code; however, the function calls must be structured such that they can easily accommodate new and different types of devices, as well as be used from any one of numerous programming languages.

3 The Operating System Analogy

In many ways the integrated environment that SHELLEY requires is analogous to a modern operating system. An operating system serves as an interface between the user-level software on a computer and the computer's hardware [3]. It has two primary tasks: provide convenience for the programs running on the computer and do so efficiently. The modern operating system can also be considered a control program that manages system resources (memory and processing time, as well as system Input/Output devices) by resolving conflicting resource requests and guaranteeing effective use of these limited resources. The operating system also controls application processes and ensures that the system is properly used by these processes [3].

One model of operating system is the one program running at all times on the computer – generally called the *kernel.* Following this, all other processes are applications that provide some functionality, either for the kernel, or for the user. It is the kernel that processes all system calls, handles all sharing of central processing unit (CPU) time and random access memory (RAM) between competing processes, and performs the handling of peripherals. To access peripherals, processes must do so through the kernel (see **Figure 1**) [4]. Another crucial part of an operating system is the capacity for multiprocess scheduling and management – a significant aspect of the modern operating system, and one that is crucial for our use of the model as an environment for artificial intelligence.

Given this formal model of an operating system, it can be used as a point of departure to construct our own model of an integrated environment for artificial intelligence. There are several differences, though, which merit attention before proceeding. The first of these differences is that when addressing the potential paradox of convenience and efficiency, operating system design has historically favored efficiency over convenience when the two have contradicted each other. In our model, however, though we emphasize both, convenience receives precedence when conflicts require resolution. As stated earlier, the primary goal of this work is to make more convenient the work of future users of this environment.

The second major difference is actually an issue of implementation that will be addressed later. For now, let it suffice to say that we employ a client-server model to facilitate communication, rather than use a method similar to that of system calls to the kernel. In a client-server model, we have two classifications of programs: clients, which make requests, and servers, which service the clients [5]. The client-server model may not be quite as efficient, but it certainly increases the convenience of our model. Thus, we see the influence of our favoritism for convenience over

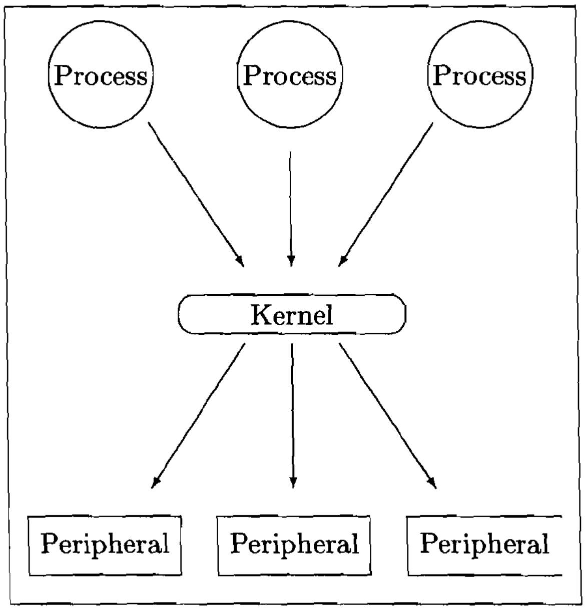

Figure 1: PROCESS/PERIPHERAL COMMUNICATION VIA THE KERNEL [4]

efficiency. The client-server model allows for network-ability, easier implementation, and allows this integrated environment to be built on top of, not replace, the existing operating system.

4 The SHELLEY Integrated Environment (SIE)

The operating system model discussed above (see **Figure 1**) offers the additional benefit of being completely modular: distinct functional units are separated into disjoint and independent components. These components then can be used to construct more complex structures, which in turn can be used as the components to build even more complex and powerful entities. This modularity provides a very simple and convenient way for developers to construct powerful systems by using the work of previous developers as building blocks. Additional advantages are that this form is very flexible and expandable, and that it facilitates program code reuse. It is with this model in mind that we have designed the SHELLEY INTEGRATED ENVIRONMENT (SIE).

4.1 The Agent/Administrator/Device Model

We cannot, however, have complete freedom in the modularity of SIE; there must be some constraints to define the relationship between these modules. Thus, we employ the operating system model as the basis for a similar structure to govern SIE: the agent/administrator/device model.

In this modular all-encompassing structure, there are three primary types of components: *agents*, which are programs for a specific task, *devices* which the agents use to accomplish these tasks, and the *administrator*, which intercedes between the other two modules, facilitating communication and regulating agent access to devices.

4.1.1 Intelligent Agents

As discussed earlier, *intelligent agents* are objects designed for a specific artificial intelligence task, such as navigating through a maze, playing chess, or performing speech recognition. Thus, to build an artificially intelligent entity, multiple agents would be run to accomplish each different behavior desired. In SIE, the intelligent agent is considered to be a software application written to accomplish a particular goal, following the general definition presented earlier of an agent as a mechanism mapping input from sensors to behavior through effectors. Agents in SIE are analogous to application processes in the operating system model.

4.1.2 Devices Modules

The agents, however, should not have to know the details of the resources (sensors and effectors) that they necessarily must use. Therefore, we call upon the *device module* to act as an interface between the agent requests and the physical robotic hardware. Device modules within the context of SIE encapsulate individual functional units. They usually control external hardware peripherals such as arms and cameras, though a device can just as easily contain a cognitive module such as an artificial neural network. Through device modules we are able to extract the implementation details of these functional units from the role of the agent programmer. These device modules are equivalent to an operating system's device drivers in that both the SIE device modules and the operating system's device drivers are software applications which facilitate the use of specific hardware by other software applications.

4.1.3 The Administrator

The challenge remaining is to integrate the agents and devices into a cohesive whole, allowing intelligent agents to use devices while still maintaining a relative degree of simplicity for the individual implementation of an agent. The solution is the *administrator* module which serves as a mediator between the agents and the devices that the agents use, much like the operating system kernel serves as a mediator between software applications and the hardware they must use. The administrator resolves all conflicts between multiple agents trying to control the same device. For example, if multiple agents require access to a robotic arm, but wish to move the arm in differing directions, it is up to the administrator to resolve this dispute. When an agent requires data from a device, the agent sends the request to the administrator who passes the request on to the device if the administrator deems the request admissible. When the device sends back data, the data is streamed to the administrator who channels it to the appropriate agent. In this way the complications of resource management are extracted from the agents and devices, and handled only by the administrator (see **Figure 2**).

4.2 Agent Ownership of Devices

Some devices such as one which interfaces with motor-driven wheels should only be controlled by one agent at any one time. Other devices such as one which acquires single frames of video from a camera may be used by multiple agents, but only one agent should have permission to make status changes to the device, for example, change the resolution or filtering mode in the case of a video frame acquisition device module. Given these stipulations, the administrator employs a mechanism for read/write permissions similar to that of a Unix operating system. If a device is designated as "sharable" then it can have an unlimited number of agents using it, though it can have at most one owner with full read/write permission at any one time (ownership equals write

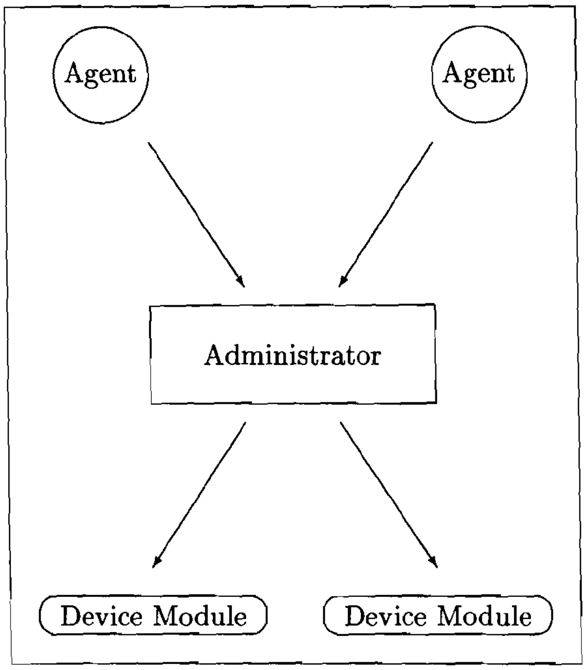

FIGURE 2: THE SIE AGENT/ADMINISTRATOR/DEVICE MODEL

permission); all other agents must use the device in read-only mode. The requests of a device which are considered read-only and those which require write permission must be explicitly made known to the administrator through a `.conf` file for each device (see **Section 4.4 The device_list File**).

This permission system introduces the additional complexity of determining which agent owns (has write permission for) a device. The administrator grants owner privileges to an agent for a specific device if the agent requests the device and no other currently connected agents own the device – either no other connected agents have requested the device, and therefore it is not yet connected, or other agent(s) are using the device, but the previous owner has relinquished ownership and no other agent in the interim has requested ownership.

Devices may also be specified to allow multiple instances. This mostly like would occur with cognitive modules, or at least devices which operate completely at a software level and do not interface with external peripherals. If an agent requests a device of this type, then each request will result in a new device of that type to be run. Therefore, an agent who requests a device of this type is guaranteed to be granted owner permissions because it is the only agent using that instance of the device.

To further facilitate sharing of devices and inter-agent cooperation, an agent can query the current ownership status of a device and receive a response of either (a) the querying agent owns the device, (b) another agent owns the device, (c) no agent owns the device, or (d) the device is not known. The agent can also make a request to the administrator to claim ownership of a device, receiving either a confirmation or rejection. In the case of a rejection, the administrator sets a flag which indicates that there are agents without ownership who desire ownership. Agents can query the status of this flag to know if other agents have been requesting ownership. Additionally, the agent can relinquish ownership of a device. Through this system, multiple agents can effectively share a device by yielding ownership when they are able to operate in read-only mode, and by requesting ownership only when it is absolutely necessary to have write permission. Of course, handling of ownership issues is not necessary for an agent implementation; if an agent is not intended for use alongside other agents, then behaving in a "device-greedy" manner is completely acceptable.

4.3 The Flow of SIE

Just as the kernel runs the entire time the operating system is running on a computer, our administrator module runs as a background process whenever we are using SIE. If no intelligent agents are running, then the administrator simply waits, listening for agents. When an agent is run, it connects to the administrator, informing it of what devices are needed. After each device request, the agent listens for a response from the administrator, who processes the device request and makes one of the following responses if an error occurs:

- The requested device is currently in use by another agent, and the device is not sharable. The administrator gives the agent the choice of either exiting or continuing without the device.
- The requested device is currently in use by another agent, but the device is sharable. Thus, the agent can access the device, but will not have ownership permissions. The administrator gives the agent the choice of either exiting or continuing without ownership of the device.
- The requested device is unknown to the administrator. The administrator gives the agent the choice of either exiting or continuing without the device.

If a device is successfully connected to the administrator with the requesting agent as owner, then the administrator sends a confirmation to the agent. After each confirmation by the administrator (or acceptance of restrictions by the agent) the administrator sends the agent a unique identification number which the agent and administrator then use to refer to the device in all future communications. Note that this device identification number is not the same as the identification number listed in the `device_list` file. This identification number serves the purpose of allowing the agent and administrator to refer to a specific device, distinguishing between instances of the same device type. If an instance of a device is shared between multiple agents, then the same number is used by all agents to refer to this device. For example, if there is a video camera device module, an agent may require two instances of this module (one for each of SHELLEY's two cameras). Thus each instance of the module would be referred to by a different identification number so that the agent and administrator can distinguish between the two. Also, if multiple agents are sharing the same video camera module, then both agents use the same number to refer to the same device.

Such different situations may appear to make initialization of an agent overly complicated, but an agent could be written very simply by connecting to the administrator, asking for certain devices, and immediately failing if confirmation is not received. Depending on the context in which this agent will be run, this approach may be sufficient. However, in cases where the programmer wishes to maximize the stability of the agent and build support for inter-agent cooperation, handling of the above situations is necessary, along with the ownership issues discussed previously.

When an agent disconnects from the administrator, any devices used exclusively by that agent are also disconnected by the administrator. If the device is being used by other agents, then the device will remain until all agents accessing it disconnect.

4.4 The device_list File

When the administrator is started, it reads a `device_list` file that lists all the device modules that will be supported by the administrator. The `device_list` file must adhere to the following syntax: lines that begin with white space or pound signs ("#") will be ignored; lines which define a device must have the following information in order (separated by white space):

- A positive integer, to be used as a unique identification number for the device. This number is used both internally by the administrator and by agents referring to the device at time of request.
- The device executable name (with full path).
- The host computer on which the device should be run. This currently has no effect – all device modules are run on the host of the administrator (see **Section 7 Future Work** for a discussion of running devices on remote computers).
- Either `share` or `no-share` to designate if the device can be shared by multiple agents.
- An integer to designate the maximum number of agents which can share the device. If the number is 0, then there is no fixed maximum. If the device is designated as not shareable, then this value must still be here, but it serves no functional purpose.
- Either `multiple` or `no-multiple` to designate if the device can have only one instance or multiple instances.
- An integer to designate the maximum number of instances which can exist for that device. If the number is 0, then there is no fixed maximum. If the device is designated as `no-multiple`, then this value must still be here, though it is meaningless.

Note that `multiple` overrides `share`; if a device is marked to be both `multiple` and `share`, a separate instance of the device will be created at each request – the device will never be shared until the maximum number of instances for that device is achieved, at which point the device will be shared.

Below is a sample `device_list` file. The lines with pound signs are comments. Each other line specifies a device by executable name and unique id number by which both administrator and agents will refer to the device.

```
# Andy Ritger
# 4-12-99
# Research Honors
# sample device list file
5 /opt/local/shelley/devices/frame_grabber localhost share 0 no-multiple 0
1 /opt/local/shelley/devices/neural_net localhost no-share 0 multiple 0
17 /export/home/aritger/temp/mobot_wheels localhost no-share 0 no-multiple 0
```

The administrator also requires a file associated with the device in the same directory as the executable called `<executable name>.conf` (for example: `frame_grabber.conf` or `neural_net.conf`). These `.conf` files specify which device requests, if any, are considered to require write permission. If the administrator cannot find the file, it will produce a warning and proceed under the assumption that all requests require ownership to be performed.

4.5 Network Protocols and the Details of Inter-Module Communication in SIE

4.5.1 Choice of Communication Medium

The mechanism for inter-module communication within SIE is the *sockets Application Programmer's Interface* (API), following a client-server model where the administrator functions as the server, and the agents and devices function as clients. Sockets were chosen over other forms of interprocess communication (IPC), such as shared memory, pipes, and signals [6], because they are built on top of TCP/IP, and therefore facilitate the possibility of networking and distributing module execution over the different computers that comprise SHELLEY.† Sockets are a sequenced, reliable, fast, bidirectional means of interprocess communication through variable length streams [4]. The sockets are of type `SOCK_STREAM` and domain `AF_INET`, which allows the client-server to connect and communicate anywhere on the Internet [7].

To ease future implementation, several C wrapper functions are provided for simple socket creation and use (`shelley_sockets.h` and `shelley_sockets.c`; see **Appendix C** and **Appendix D**, respectively). Future contributors are not bound to use the SHELLEY_SOCKETS mini-library, but the basic facilities are provided and a complete – though rudimentary – client-server structure can be built solely with SHELLEY_SOCKETS function calls. **Figure 3** gives a detailed chart of the layers of protocol used in SIE.

4.5.2 Defining a Communications Protocol

In his 1997 paper, Douglas Gage discusses the obstacles he has encountered in networking mobile robot systems [7]. His approach is primarily for defense purposes using a wireless RF networking system, which differs from this project in that we do not have the same constraints of long bandwidth-delay, error prone links, and of being mission critical. Nevertheless, his discussion on communication protocol is still very much applicable. Gage defines protocol as: "a language used by two entities to exchange information over a communications channel – it represents a shared understanding or agreement of how each entity will interpret the signals it receives from the other" [7]. To facilitate communication between the three types of modules in SIE, we must therefore intricately define the signals that will be sent between the modules.

SIE's communications protocol is as follows: once the socket connection has been established, 1-byte messages are sent across the connection, always initiating at the agent in agent-administrator communication, and at the administrator in administrator-device module communication.‡ This

†While SHELLEY currently consists of one Sun Ultra I Workstation and three Intel-based personal computers, there is no reason why that could not change in the future, and we can maintain the flexibility emphasis in our design by allowing for any number and type of computers.

‡SIE employs its own defined types `int8` (one byte, unsigned) and `int32` (four bytes, unsigned) for all its communication. This is important for portability. If SIE is ported to a platform with a different sized integer (a different number of bits), the only change which needs to occur is the definition of SIE's `int8` and `int32` [8]. The `int8` and `int32` type definitions are in the `sie_protocol.h` file; see **Appendix A**.

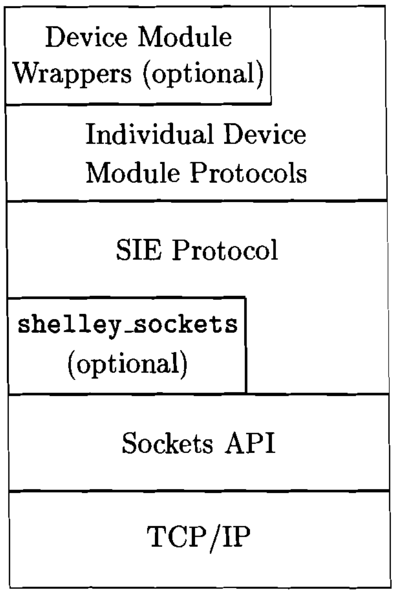

Figure 3: The Layers of Inter-Module Communication Protocol used in SIE

forces the condition that the administrator cannot directly broadcast information to agents; if something changes at the administrator, such as another agent relinquishing device ownership, agents can only find out this information by explicitly requesting it. Similarly, at the connection between the administrator and a device, state changes at the device can only be known by the administrator if it explicitly queries the device. This may at first appear confining, but it greatly simplifies the protocol between any two devices, because both will always know which is expected to send the next message.

When an agent connects to the administrator, there is a sequence of startup information passed back and forth in the following format: the agent sends the `AGENT_CONNECT` message†, to which the administrator then replies with the `ADMIN_ACKNOWLEDGE_AGENT_CONNECT` message. Next, for each device the agent requires, it sends `AGENT_DEVICE_REQUEST` followed by an `int32` integer which is the device identification number specified in the file `device_list`. The administrator then responds with `ADMIN_CONFIRM_DEVICE_REQUEST` if the device was successfully connected, and owner privileges granted to the requesting agent, otherwise, one of the following errors is sent by the administrator: `ADMIN_DEVICE_UNKNOWN`, `ADMIN_DEVICE_ALREADY_OWNED`, or `ADMIN_DEVICE_NOT_AVAILABLE`. In all three error cases the agent has the option of accepting the error (`AGENT_ACCEPT`) and the stipulations which that implies (see **Section 4.3 The Flow of SIE**) or failing (`AGENT_FAIL`), in which case there is no further communication between the administrator and the agent; the administrator disconnects the agent and frees all resources used exclusively by that agent. After the administrator has sent the `ADMIN_CONFIRM_DEVICE_REQUEST`, the administrator also sends a 32-bit integer which is a unique number which the agent should then use whenever referring to the device. The number is also sent after the agent sends the `AGENT_ACCEPT` message. When all devices have been requested and either confirmed, or errors accepted, the agent sends `AGENT_DEVICE_REQUEST_DONE`, indicating the end of startup communication between the agent and the administrator.

†All messages are declared as constants through C `#define` statements in the `sie_protocol.h` header file, see **Appendix A**.

When a device is requested, the administrator uses the device id number given by the agent to lookup the device executable (this information is stored in the device list file) and run it, following the convention "<device executable>␣<computer hosting the administrator>␣<port number on which the administrator is listening for devices>." For example:

```
/opt/local/shelley/devices/frame_grabber localhost 4096
```

The administrator then waits for the device to connect to it, sending an acknowledgement upon connection, `ADMIN_QUERY_DEVICE`, to which the device responds with either `DEVICE_READY` or `DEVICE_FAILED` if the device module experienced some internal error and was not able to acquire all its needed resources. On a `DEVICE_FAILED`, the administrator sends an `ADMIN_DISCONNECT_DEVICE` message to the device allowing it to exit cleanly, and informs the requesting agent that the device is unavailable.

After these initial exchanges of startup information between agents and the administrator, and devices and the administrator, the specific protocol for a device must be explicitly defined for every device type. The administrator examines the device's `.conf` file to know which device requests require write permissions (requests not listed in the `.conf` file are assumed to only require read permission). However, beyond knowledge of what requests can only be issued by the owner, the administrator does not need to know any more specifics of the device protocol, and merely channels allowable requests through from agent to intended device, and from device to appropriate agent.

For an agent to send a command to a device, the agent sends the message `AGENT_SEND_DEVICE` followed by two `int32` numbers: the unique identifying integer to specify the device, and the length (in bytes) of what is being sent to the device. There is the further stipulation that the first byte of the message for the device must be the request code. If the agent is not owner, the administrator compares this request with the requests listed in the `.conf` file for the device in question, and determines if the message can be sent to the device. If the administrator determines that the message can be sent to the device, the message (stripped of the `AGENT_SEND_DEVICE`, the `int32` device id number, and the `int32` message length). The administrator also notes which agent sent the message, so that when the device responds, the message can be channelled to the correct agent. The two requirements we place on device protocol in SIE are: (1) devices must always send some response back after receiving a request, and (2) the device must prepend this response with an `int32` indicating the size (in bytes) of the response.

Finally, the agent can send the `AGENT_DISCONNECT` message which tells the administrator that the agent is quitting. At this point the administrator assumes that it will receive no more communication from the agent, and frees any resources that had been allocated for the agent. If the disconnecting agent is the only agent using any devices, those devices are sent the `ADMIN_DISCONNECT_DEVICE` message.

5 How SIE Can Support Different Paradigms of Artificial Intelligence

The agent module is defined no further than the communication protocols through which an agent talks to devices in order to sense and react to its surroundings. Thus, flexibility is built into the foundation of the SIE structure, allowing support for any approach to the agent design. The *classical* approach to robotics and artificial intelligence is to construct an internal model of the world upon which we have our agents make decisions. All sensory inputs are gathered together and information conflicts are resolved to construct a consistent world view [9]. The advantage to this approach is that all information about the world is centralized and there is one single entity which is fed all the information and can therefore make the most complete and well-informed decision about how to react. This paradigm is very well supported by our integrated environment. The simplest implementation would be to have a single agent that requested from the administrator all the necessary devices to build a world model. The agent could then make calls to all the devices to request data, receive that data, construct a model of the world, and make a decision about how to behave.

Rodney Brooks, of the MIT Artificial Intelligence Laboratory, however, condemns this traditional approach because it is slow (computationally intensive to build a world model) and large (much memory is required to store the internal model). Brooks instead advocates a *subsumption architecture* wherein the decisions are not made by a single agent, but is distributed over an organized hierarchy of behavior modules that directly map perception to action [10]. These separate behavior modules do not directly communicate with each other in making decisions, but rather inhibit other modules when they are active. For example, there may be a behavior that tells a mobile robot to continuously move forward, but there may be another behavior that tells the robot to stop if there is an obstacle in its path. If, in our hierarchy, we defined that the stopping behavior inhibits the forward behavior, our robot will travel forward (the stopping behavior has no reason to be active, and therefore lies dormant and does not inhibit the forward behavior) until the robot encounters an obstacle. When an obstacle is encountered, the stopping behavior is made active, which inhibits the forward behavior – the robot comes to a stop. By defining a hierarchy of behaviors and by defining how these behaviors interact and inhibit each other, we can construct a system that does not need a central intelligence, but can behaved based on a series of smaller intelligences. SIE can easily support this approach to designing intelligent agents by building a separate agent for each "behavior module" and having them access the needed devices through the administrator. Agents – in this case functioning as behavioral modules – can inhibit each other in the way Brooks prescribes using the SHELLEY_SOCKETS mini-library to produce direct inter-agent communication.

6 An Example of SIE Applied: Identifying a User

A simple example of SIE in use is to address the task of recognizing the person sitting at SHELLEY's console. The task is accomplished by constructing one agent that accesses two devices: a frame grabber module that upon request returns a frame of video from the Sun video cameras used by SHELLEY, and an artificial neural network module that encapsulates all the functionality and data structures of a neural network, complete with facilities for training using the backpropogation algorithm [11]. This presents an example of a device module which does not interface with any external physical hardware. Instead, the artificial neural network device module provides a completely self-contained functional unit. It is still valid, however, to consider this as a device module because all any device module does is provide some function which should be separate and distinct from the role of the agent.

The user-identifier agent connects to the administrator, and requests both the frame grabber and artificial neural network devices, failing if owner permission cannot be granted for both. After initialization, the agent presents to the current user a menu with options: "identify user," "capture frames to pgm," "train," or "quit." Through this user interface, we can acquire a series of video frames and save them as pgm image files, use the image files to train the network, and then test the network with live video from one of SHELLEY's Sun video cameras. This functionality demonstrates the use of multiple devices by a single agent, all communicating through the administrator. The specifics of the frame grabber and neural network device protocols can be found in `frame_grabber_protocol.h` (see **Appendix E**) and `neural_net_protocol.h` (see **Appendix I**), respectively.

This example demonstrates several key features of SIE. First, SIE's flexibility is exhibited in the implementation styles employed. While the administrator is implemented in strict C (primarily to support multi-threading), the user-identifier agent is implemented in C++. In reality, the devices and agents can be individually implemented in any language with bindings to the sockets API. As long as the communications protocol is followed, one module does not need to know the implementation details of any other module.

Another example of SIE's abstraction of implementation details is the way in which external hardware can be easily changed without disrupting SIE. If the current Sun video cameras were replaced by different peripherals for visual perception, then all that would be needed would be a new frame grabber module that used the same communications protocol for the previously existing agents to still be useful. This system allows agents to not be concerned with the specifics of the neural network itself (see `administrator.c` in **Appendix B** and `agent.C` in **Appendix M**).

Finally, the largest single benefit of SIE is the ease of implementation of the user-identifier agent. Simplicity was further increased by the development of C++ wrapper classes to encapsulate the communication with each device; when the agent creates an instance of each wrapper class, the constructor handles the startup communication with the administrator. The agent then calls methods of the classes to send all the requests to the devices (via the administrator) and collect the resulting data (see `Frame_Grabber.H` in **Appendix G**, `Frame_Grabber.C` in **Appendix H**, `Neural_Net.H` in **Appendix K**, and `Neural_Net.C` in **Appendix L**). These classes are declared and defined in separate files from the user-identifier so that they can be used by other agents. Of course, the implementation of future agents that use these same devices need not employ the wrapper classes, and can rely directly on the protocol header files.

7 Future Work

This research thus far has focused primarily on the design of (SIE), with selective implementation of key points to test theories, verify strategies, and prove concepts. With the network and communication protocols established, as well as the overall flow of SIE well defined, the next step is implementation. The administrator module exists in skeletal form, and while the majority of the central issues are addressed, the multiple agent support and write permission functionality, though well defined, is yet to be implemented.

The following is a partial list of some additional future contributions which could be made to SIE:

- **Network-ability:** Currently, agents can connect to the administrator if it is running on the localhost, or if it is running on any other computer networked to the host of the agent.

However, because the administrator must create the device processes, there is currently no means for the devices to be run on a computer other than the one upon which the administrator is being run. Perhaps an investigation of an RPC (remote procedure call) package may provide a solution [6]. An alternative answer could be to not have devices get executed by the administrator when they are needed, but rather run them explicitly and connect them to the administrator at administrator startup, leaving them connected for the entire duration of SIE. In this way, the user could explicitly run the device module from any networked computer, though an obvious disadvantage would be that it would then become a responsibility of the user to ensure that all needed devices were running and connected. Or, perhaps a separate administrator could be run on each computer of a networked cluster, so that when one administrator needed to access resources on another computer, it could be done through inter-administrator cooperation. The advantages to distributing SIE over a networked cluster are many, as are the issues involved which would require addressing.

- **Varying Agent Priorities:** A priority scheme for agent ownership of devices maybe useful for SIE in situations where many agents are being run concurrently. This however, creates the added complexity of communicating to a "less important" agent that a more important agent has come along and ruthlessly usurped device ownership.

- **Midprocess Device Requests:** The present design of SIE forces all device requests by an agent to occur when the agent initially connects to the administrator. A potentially very useful modification could be a design for agents to request devices anytime during their session with the administrator, and not only upon connection.

- **Device "Short-circuiting:"** In the case where an agent essentially streams the data coming in from one device to another device (sending image data from a video frame grabber to an artificial neural network, for example) there would be a performance increase gained if the agent could tell the administrator to channel the data to a specific device rather than send it back to the agent. Very quickly, complications arise when considering this scheme due to the implicit need this creates for differing devices to have compatible communication protocols, which is otherwise not an issue in the current design of SIE.

- **Development of Agents and Devices:** Perhaps the most obvious contribution to be made to SIE is the development of both agents and devices. Ideally, device modules can be created, along with a defined protocol for accessing their functionality, and then saved – building a library of devices which can then be used by agents as they are created. Most likely, device module development will be driven by necessity – when an agent requires access either to a peripheral or some distinct functional unit for which there is no current device module written. It is hoped that the design philosophy of building separate, reusable modules will be followed to maximize code reuse and long-term productivity.

8 Conclusion

The SHELLEY Integrated Environment (SIE) is designed primarily with the goal of easing implementation of future projects by providing an easy means for accessing the devices which allow SHELLEY to interact with her surroundings. The design emphasizes flexibility and expandability, as well as simple code reuse in the form of separate modules. The agent/administrator/device model upon which SIE is built allows the implementors of agents to not be concerned with the inner workings of accessing specialized hardware – this is localized to specific device modules. Inter-module communication is accomplished using the sockets API, which offers the future opportunity to distribute SIE over a network of computers. Multiple agents can be run in conjunction, building an integrated system of behaviors. It is the administrator's responsibility to regulate and manage agent access to devices, much like in the modern operating system, it is the kernel's responsibility to regulate and manage processes and their access to system resources. Finally, the specifics of how SHELLEY maps sensory input to behavioral output is encapsulated in the agents, thus SIE serves only to facilitate and does not confine how future researchers approach the problem of building an artificially intelligent entity.

Appendix A sie_protocol.h[†]

```
/*************************************************************************
  Andy Ritger
  Research Honors
  4-26-99

  The constants which comprise the SIE protocol follow the simple naming
  convention where the first word is either AGENT, DEVICE, or ADMIN to
  designate who is sending the message, followed by an underscore separated
  description of the message.

  Note that all constants are sent across sockets as type int8.

  *************************************************************************/



#ifndef ___SIE_PROTOCOL___
#define ___SIE_PROTOCOL___



/*************************************************************************
  When porting SIE to other platforms, edit these typedefs as needed so that
  byte8 is an unsigned 8-bit value, and int32 is an unsigned 32 bit value.
  *************************************************************************/

typedef unsigned char int8;
typedef unsigned int int32;



/*************************************************************************
  The AGENT_CONNECT message is sent by the agent after a socket connection
  has been established.  The agent then awaits confirmation from the
  administrator.
  *************************************************************************/

#define AGENT_CONNECT 32



/*************************************************************************
```

†All of the source code listed in these appendices can be found at www.iwu.edu/~shelley/sie

```
   The ADMIN_ACKNOWLEDGE_AGENT_CONNECT is sent by the administrator to an
   agent after the agent has sent the AGENT_CONNECT message.  This is simply
   a means of "handshaking" so that one can verify the other's existence.
   ****************************************************************************/

#define ADMIN_ACKNOWLEDGE_AGENT_CONNECT 33



/****************************************************************************
   The AGENT_DEVICE_REQUEST message is sent by the agent to the administrator
   after the ADMIN_ACKNOWLEDGE_AGENT_CONNECT is received.  The
   AGENT_DEVICE_REQUEST is followed by an int32 which specifies the id number
   for a device as given in the file device_list.
   ****************************************************************************/

#define AGENT_DEVICE_REQUEST 34



/****************************************************************************
   The ADMIN_CONFIRM_DEVICE_REQUEST is sent by the administrator to the agent
   to confirm that the requested device has been verified and the agent granted
   ownership.  This message is followed by an int32 which is the unique
   identifying number of the specific instance of the device module, which
   the agent and administrator will use for all future communication regarding
   the device module.
   ****************************************************************************/

#define ADMIN_CONFIRM_DEVICE_REQUEST 35



/****************************************************************************
   The ADMIN_DEVICE_UNKNOWN is sent by the administrator to the agent when
   the requested device id is unknown to the administrator (the given id
   number is not listed in the device_list configuration file.  This message
   is followed by an int32 which is the unique identifying number of the
   specific instance of the device module, which the agent and administrator
   will use for all future communication regarding the device module.  This
   number is not really needed, but is given to conform with the conventions
   followed for other other possible responses made by the administrator
   regarding device module requests.
   ****************************************************************************/

#define ADMIN_DEVICE_UNKNOWN 36
```

```
/***************************************************************************
  The ADMIN_DEVICE_ALREADY_OWNED message is sent by the administrator to the
  agent in response to a device module request if the device module exists,
  but is already owned by another agent (thus write permission cannot be
  granted to the requesting agent).  This message is followed by an int32
  which is the unique identifying number of the specific instance of the
  device module, which the agent and administrator will use for all future
  communication regarding the device module.
  **************************************************************************/

#define ADMIN_DEVICE_ALREADY_OWNED 37



/***************************************************************************
  The ADMIN_DEVICE_NOT_AVAILABLE message is sent by the administrator to the
  agent in response to a device module request if the device module is known
  by the administrator, but it is not available -- either the maximum number
  of agents are already using it, or there was an error when the administrator
  attempted to create the device.  This message is followed by an int32 which
  is the unique identifying number of the specific instance of the device
  module, which the agent and administrator will use for all future
  communication regarding the device module.
  **************************************************************************/

#define ADMIN_DEVICE_NOT_AVAILABLE 38



/***************************************************************************
  The AGENT_ACCEPT message is sent by the agent to the administrator after
  one of the above three error messages have been sent (ADMIN_DEVICE_UNKNOWN,
  ADMIN_DEVICE_ALREADY_OWNED, or ADMIN_DEVICE_NOT_AVAILABLE) to specify
  that the conditions imposed by the given error will be accepted and the
  agent wishes to continue.
  **************************************************************************/

#define AGENT_ACCEPT 39



/***************************************************************************
  The AGENT_FAIL message is sent by the agent to the administrator after
  one of the above three error messages have been sent (ADMIN_DEVICE_UNKNOWN,
  ADMIN_DEVICE_ALREADY_OWNED, or ADMIN_DEVICE_NOT_AVAILABLE) to specify
  that the conditions imposed by the given error will not be accepted and the
  agent wishes to fail without proceeding further.  As soon as this message
```

```
  is received, the administrator assumes that the agent is gone, and ignores
  its existence.
  ****************************************************************************/

#define AGENT_FAIL 40



/****************************************************************************
  The AGENT_DEVICE_REQUEST_DONE message is send by the agent after it has
  requested all necessary device modules and has dealt with the administrator's
  responses.
  ****************************************************************************/

#define AGENT_DEVICE_REQUEST_DONE 41



/****************************************************************************
  The ADMIN_QUERY_DEVICE message is sent to the device by the administrator
  after the device has connected to ensure that the device really is a device.
  ****************************************************************************/

#define ADMIN_QUERY_DEVICE 42



/****************************************************************************
  The DEVICE_READY message is sent by the device to the administrator in
  response to the administrator's ADMIN_QUERY_DEVICE.  The message indicates
  that the device is ready to receive commands.
  ****************************************************************************/

#define DEVICE_READY 43



/****************************************************************************
  The DEVICE_FAILED message is sent by the device to the administrator in
  response to the administrator's ADMIN_QUERY_DEVICE.  The message indicates
  that the device experienced some internal error and is not able to function.
  The administrator assumes that the device module goes away after this message
  is sent
  ****************************************************************************/

#define DEVICE_FAILED 44
```

```
/***************************************************************************
  The ADMIN_DISCONNECT_DEVICE message is sent by the administrator to a
  device module to tell the device that it is no longer needed and should
  exit.
  ***************************************************************************/

#define ADMIN_DISCONNECT_DEVICE 45



/***************************************************************************
  The AGENT_QUERY_DEVICE_OWNERSHIP message is sent by an agent to the
  administrator.  The message is immediately followed by an int32 holding
  the specific identification number of a device module.  This is used
  by an agent to query the ownership of a device module.
  ***************************************************************************/

#define AGENT_QUERY_DEVICE_OWNERSHIP 46



/***************************************************************************
  The ADMIN_THIS_AGENT_OWNS_DEVICE is sent by the administrator to an agent
  is response to the agent's AGENT_QUERY_DEVICE_OWNERSHIP message.  This
  indicates that the requesting agent is owner of the device module in question
  (the agent has write permission).
  ***************************************************************************/

#define ADMIN_THIS_AGENT_OWNS_DEVICE 47



/***************************************************************************
  The ADMIN_ANOTHER_AGENT_OWNS_DEVICE is sent by the administrator to an agent
  is response to the agent's AGENT_QUERY_DEVICE_OWNERSHIP message.  This
  indicates that an agent other than the requesting agent is the the owner
  of the device in question (thus the requesting agent only has read
  permission).
  ***************************************************************************/

#define ADMIN_ANOTHER_AGENT_OWNS_DEVICE 48



/***************************************************************************
  The ADMIN_NO_AGENT_OWNS_DEVICE message is sent by the administrator to an
  agent is response to the agent's AGENT_QUERY_DEVICE_OWNERSHIP message.  This
```

```
  indicates that no agent owns the device in question (no agent has write
  permission).
  ****************************************************************************/

#define ADMIN_NO_AGENT_OWNS_DEVICE 49



/****************************************************************************
  The AGENT_REQUEST_DEVICE_OWNERSHIP message is sent by an agent to the
  administrator. The message is immediately followed by an int32 holding
  the specific identification number of a device module.  This is used
  by an agent to request that it be made owner of the device in question.
  ****************************************************************************/

#define AGENT_REQUEST_DEVICE_OWNERSHIP 50



/****************************************************************************
  The ADMIN_GRANT_DEVICE_OWNERSHIP message is sent the administrator to
  an agent in response to the agent's AGENT_REQUEST_DEVICE_OWNERSHIP
  message.  This indicates that the agent now is owner (has write permission)
  for the device module in question.
  ****************************************************************************/

#define ADMIN_GRANT_DEVICE_OWNERSHIP 51



/****************************************************************************
  The ADMIN_DENY_DEVICE_OWNERSHIP message is sent by the administrator to
  an agent in response to the agent's AGENT_REQUEST_DEVICE_OWNERSHIP message.
  This indicates that the agent does not receive ownership (given write
  permission) for the device module in question.
  ****************************************************************************/

#define ADMIN_DENY_DEVICE_OWNERSHIP 52



/****************************************************************************
  The AGENT_SEND_DEVICE message is sent by an agent to the administrator to
  indicate that what is following is a message for a device.  It is followed
  by two int32s; the first indicates the id number of the device to which
  the message should be sent, and the second specifies the length of the
  message to be sent to the device module.
  ****************************************************************************/
```

```
#define AGENT_SEND_DEVICE 53



/****************************************************************************
  The AGENT_DISCONNECT message is sent by an agent to the administrator to
  indicate that the agent is done and is exiting.  Upon receiving this message,
  the administrator releases any resources used exclusively by the
  disconnecting agent, and hereafter assumes that the agent no longer exists.
 ***************************************************************************/

#define AGENT_DISCONNECT 54



#endif
```

Appendix B administrator.c

```
/******************************************************************************
  Andy Ritger
  Research Honors
  4-26-99

  administrator.c

  This is the administrator source code.  The administrator is still in *VERY*
  rudimentary form, but the basic functionality is here.

  ******************************************************************************/



#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <fcntl.h>
#include <ctype.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/socket.h>

#include "sie_protocol.h"



/******************************************************************************
  structures
  ******************************************************************************/

typedef struct device_info
{
  device_info *next;  /* pointer so that we can have this in a linked list */
  int dev_id;         /* the device id as specified in the device_list file */
  char *path;         /* path and name of binary of device module */
  char *bin;          /* file name of binary */
  char *host;         /* host on which to run device module (not implemented)*/
  int shared;         /* 1 = device can be shared; 0 = not */
```

```
  int multiple;       /* 1 = device can have multiple instances; 0 = not */
};



/******************************************************************************
  prototypes
  ******************************************************************************/



/******************************************************************************
  The function agent_listener () binds a socket, and listens for an agent
  connecting on that socket.  After this happens, we attempt to acquire
  all the device modules requested by the connecting agent.  Finally, we
  sit in an loop and pass device commands from the agent to the appropriate
  device, and then from the device back to the agent.
  ******************************************************************************/

void agent_listener (int port_number);



/******************************************************************************
  The connect_device () function is called for each device requested.  A
  child process is forked off to exec the device module program.  We then
  listen for the device to connect back to us.
  ******************************************************************************/

int connect_device (int n);



/******************************************************************************
  The read_device_list () function parses the device_list file, and stores
  all the relevant information in a linked list.
  ******************************************************************************/

device_info *read_device_list ();



/******************************************************************************
  The function find_next_data () takes a file handler and moves the handler
  past any whitespace or lines with pound signs ("#") so that next thing is
  valid data.
  ******************************************************************************/

```

```
void find_next_data (FILE *file);



/****************************************************************************
  global variables -- need to be changed
  ****************************************************************************/

device_info *device_list;
int the_socket;
sockaddr_in name;
int number_of_devices = 0;
int device_connections [10];



/****************************************************************************
  main ()
  ****************************************************************************/

int main (int argc, char **argv)
{
  /* read the device_list file */

  device_list = read_device_list ();

  /* listen for agents trying to connect */

  agent_listener (2048);

  /* tell all connected devices to go away */

  int8 send = ADMIN_DISCONNECT_DEVICE;
  for (int i = 0; i < number_of_devices; i++)
    write (device_connections [i], &send, sizeof (int8));

  printf ("ADMINISTRATOR: done.\n");
  exit (0);
} /* main () */



/****************************************************************************
  agent_listener ()
  ****************************************************************************/

void agent_listener (int port_number)
{
```

```
  int *temp, response, i, n;
  int ns, player_no;
  int len, res;
  char response2, *pname;
  int8 received, answer;

  /* create the socket */
  the_socket = socket (AF_INET, SOCK_STREAM, 0);

  /* set the port number */
  name.sin_family = AF_INET;
  name.sin_port = htons (port_number);
  n = INADDR_ANY;
  memcpy (&name.sin_addr, &n, sizeof (long));

  /* enables reuse of the port number -- this is a very good thing */
  temp = (int*) malloc (sizeof (int));
  *temp = 1;
  setsockopt (the_socket, SOL_SOCKET, SO_REUSEADDR, (char*)temp, sizeof (int));

  /* sets the size of the send and receive buffer -- 64 kb */
  *temp = 64; /* kilobytes */
  setsockopt (the_socket, SOL_SOCKET, SO_SNDBUF, (char*) temp, sizeof (int));
  setsockopt (the_socket, SOL_SOCKET, SO_RCVBUF, (char*) temp, sizeof (int));

  /* attempts to bind the socket -- failure returns -1 */
  res = bind (the_socket, (struct sockaddr*) (&name), sizeof (sockaddr_in));
  if (res == -1)
  {
    printf ("ADMINISTRATOR ERROR: unable to bind socket to port number %d.\n",
     port_number);
    return;
  }

  /* listen to the socket, waiting for agents to connect... */
  listen (the_socket, 5);
  len = sizeof (sockaddr_in);

  ns = accept (the_socket, (struct sockaddr*) (&name), &len);

  /* we've received a connection, check to see if it's an agent */
  read (ns, &received, sizeof (int8));

  if (received == AGENT_CONNECT)
  {
    printf ("ADMINISTRATOR: an agent has connected...\n");
    answer = ADMIN_ACKNOWLEDGE_AGENT_CONNECT;
    write (ns, &answer, sizeof (int8));
```

```
  }
  else
  {
    printf ("ADMINISTRATOR: something has connected on the agent port,\n");
    printf ("               but it didn't identify itself as an agent.\n");
    printf ("               Proceeding, but problems may arise.\n");
  }

  /* this will either be a device request, or a device done... */
  read (ns, &received, sizeof (int8));

  int32 val;

  while (received != AGENT_DEVICE_REQUEST_DONE)
  {
    printf ("ADMINISTRATOR: handling device request...\n");

    if (received == AGENT_DEVICE_REQUEST)
    {
      read (ns, &val, sizeof (int32));
      device_connections [number_of_devices] = connect_device (val);
      number_of_devices++;
    }

    else
    {
      printf ("ADMINISTRATOR: agent device request protocol not followed\n");
      printf ("               by connecting agent. Proceeding, but\n");
      printf ("               problems may arise.\n");
    }
    /* get next command (either a device request or a device done) */
    read (ns, &received, sizeof (int8));
  }

  printf ("ADMINISTRATOR: device module setup complete\n");

  int32 device_id, length;
  char* message;

  /* block on a read until the agent tells us to do something */
  read (ns, &received, sizeof (int8));
  while (received != AGENT_DISCONNECT)
  {
    if (received == AGENT_SEND_DEVICE)
    {
      /* which device? */
      read (ns, &device_id, sizeof (int32));

```

```
      /* how long is the message? */
      read (ns, &length, sizeof (int32));

      /* this would be where we would examine the request, and verify ownership
  (if necessary) */

      message = (char*) malloc (length);

      /* just past the message through */
      read (ns, message, length);
      write (device_connections [device_id-1], message, length);
      free (message);

      /* listen on the device socket and pass what we get back to the agent */

      /* length of the message */
      read (device_connections [device_id-1], &length, sizeof (int32));
      message = (char*) malloc (length);

      /* the message */
      read (device_connections [device_id-1], message, length);
      write (ns, message, length);
      free (message);
    }
    /* block on a read until the agent tells to do something, again */
    read (ns, &received, sizeof (int8));
  }
} /* agent_listener () */



/***************************************************************************
  connect_device ()
  ***************************************************************************/

int connect_device (int n)
{
  device_info *current_node;
  current_node = device_list;

  char *bin, *path;

  /* look for device id n in the linked list */
  while (current_node)
  {
    if (current_node->dev_id == n)
    {
      bin = current_node->bin;
```

```
        path = current_node->path;
      }
    current_node = current_node->next;
    }

    /* we assume that the dev_id is in the list... needs error trapping */

    /* fork off a process to execute the device program */
    pid_t childpid;
    if ((childpid = fork ()) == 0)
    {
      if (execl (path, bin, NULL) < 0)
      {
        printf ("ADMINISTRATOR: unable to execute %s\n", bin);
        return (-1);
      }
      exit (0);
    }

    /* set up the socket to listen...*/

    listen (the_socket, 5);
    int len = sizeof (sockaddr_in);

    int return_val = accept (the_socket, (struct sockaddr*) (&name), &len);
    int8 val = ADMIN_QUERY_DEVICE;
    write (return_val, &val, sizeof (int8));

    /* listen for a response */
    read (return_val, &val, sizeof (int8));
    if (val != DEVICE_READY)
      printf ("ADMINISTRATOR: the device is not behaving as expected...\n");

    return (return_val);
  } /* connect_device () */




  device_info *read_device_list ()
  {
    int dev, len, max_share, max_mult;
    char bin[100], host[100], share[100], mult[100];

    /* open the device list file -- it must be in our directory */
    FILE *devfile = fopen ("device_list", "r");
    if (devfile == NULL)
```

```
    {
      printf ("ADMINISTRATOR ERROR: unable to open device_list\n");
      exit (0);
    }

    find_next_data (devfile); /* skip the comments and find data */

    /* create the first node in our linked list */
    device_info *dev_list = NULL;
    device_info *current_node = NULL;
    device_info *last_node = NULL;

    dev_list = (device_info *) malloc (sizeof (dev_list)+100);
    current_node = dev_list;

    while ((feof(devfile)) == 0)
    {
      /* allocate memory for the next one in line */
      current_node->next = (device_info *) malloc (sizeof (dev_list) + 100);

      /* read the data from file */
      fscanf (devfile, "%d %s %s %s %s\n", &dev, bin, host, share, mult);

      /*    printf ("%d %s %s %s %s\n", dev, bin, host, share, mult);*/

      /* copy the device id */
      current_node->dev_id = dev;

      /* copy the binary path and name */
      len = strlen (bin);
      current_node->path = (char*) malloc (len+1);
      strncpy (current_node->path, bin, len);
      current_node->path [len] = '\0';

      /* walk backwards and get the binary itself...*/
      char ch = '\0';
      int temp_len = len;
      while (ch != '/')
      {
        temp_len--;
        ch = bin [temp_len];
      }

      temp_len++;
      current_node->bin = (char*) malloc (len - temp_len);

      for (int i = 0; i < len-temp_len; i++)
        current_node->bin [i] = bin [temp_len + i];
```

```

    current_node->bin [len-temp_len] = '\0';

    /* copy the host name */
    len = strlen (host);
    current_node->host = (char*) malloc (len+1);
    strncpy (current_node->host, host, len);
    current_node->host [len] = '\0';

    /* interpret the share */
    if (strcmp ("no-share", share) == 0)
      current_node->shared = 0;
    else if (strcmp ("share", share) == 0)
      current_node->shared = 1;
    else
    {
      printf ("ADMINISTRATOR WARNING: cannot understand \'%s\' for ", share);
      printf ("device %d\n", dev);
      current_node->shared = 0;
    }

    /* interpret the multiple */
    if (strcmp ("no-multiple", mult) == 0)
      current_node->multiple = 0;
    else if (strcmp ("multiple", mult) == 0)
      current_node->multiple = 1;
    else
    {
      printf ("ADMINISTRATOR WARNING: cannot understand \'%s\' for ", mult);
      printf ("device %d\n", dev);
      current_node->multiple = 0;
    }

    /* maintain the linked list */
    last_node = current_node;
    current_node = current_node->next;

    /* skip the comments and whitespace -- just find data */
    find_next_data (devfile);
  }
  fclose (devfile);

  free (last_node->next);
  last_node->next = NULL;

  return (dev_list);
} /* read_device_list () */

```

```
/******************************************************************************
  find_next_data ()
  ******************************************************************************/

void find_next_data (FILE *file)
{
  /* if the next character is a '#' -- the line is a comment and should be
     discarded... if the next character is whitespace, it should likewise be
     removed */

  char c;
  int loop = 1;

  char garbage [100];

  while (loop)
  {
    c = fgetc (file);

    /* if we have a pound sign, then the rest of that line is comment;
       keep reading until we find a \n */

    if (c == '#')
      while (c != '\n') c = fgetc (file);

    /* if it's a digit... then we have data */
    else if (isdigit (c))
    {
      ungetc (c, file);
      loop = 0;
    }

    /* if we hit the end of the file... break the loop */
    else if (c == EOF)
      loop = 0;
  }
} /* find_next_data () */
```

Appendix C shelley_sockets.h

```
/***************************************************************************
  Andy Ritger
  Research Honors
  4-26-99

  shelley_sockets.h

  The shelley_sockets are a small collection of functions to create a socket
  connection between two independent processes either existing on the same
  computer, or distributed over a network.

  **************************************************************************/



#ifndef ___SHELLEY_SOCKETS___
#define ___SHELLEY_SOCKETS___



#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>



/***************************************************************************
  This function should be called when we want to have our server listen for
  connections on a specific port number.  The return value is the socket
  id, or -1 if the function fails.
  **************************************************************************/

int shelley_sockets_server_listen_for_client (int port_number);



/***************************************************************************
  This function should be called when we want to have our client connect to
```

```
  an existing server.  We pass in the port on which to connect, and the
  name of the machine on which the server is being run -- set this to
  "localhost" if the server is on the same machine as the client.  Returns
  -1 if it fails to connect.
  ************************************************************************/

int shelley_sockets_client_connect_to_server (int port_number, char* hostname);



/************************************************************************
  Read from the socket; this blocks until there is length_to_read bytes
  to read at the socket.  When this returns, what was read is pointed to by
  the data pointer.
  ************************************************************************/

int shelley_sockets_read (int the_socket, char *data, int length_to_read);



/************************************************************************
  Write to the socket.  What should be written should be pointed to by the
  data pointer, and be length_to_write bytes long.
  ************************************************************************/

int shelley_sockets_write (int the_socket, char *data, int length_to_write);



#endif
```

Appendix D shelley_sockets.c

```
/*************************************************************************
  Andy Ritger
  Research Honors
  4-26-99

  shelley_sockets.c

  ***********************************************************************/



#include "shelley_sockets.h"



int shelley_sockets_server_listen_for_client (int port_number)
{
  int addr = port_number; /* won't need this */
  int *temp, response, i, n;
  int ns, player_no, the_socket;
  int len, res;
  char response2, *pname;
  sockaddr_in name;

  /********************************************************
    see the man page on socket (SunOS 5.5, socket(3N)):

    socket() creates an endpoint for communication and returns a descriptor.

    ...The domain parameter specifies a communications domain within which
    communication will take place; this selects the protocol family which
    should be used.  The protocol family generally is the same as the address
    family for the addresses supplied in later operations on the socket.
    These families are defined in the include file <sys/socket.h>...

    ...A SOCK_STREAM type provides sequenced, reliable, two-way connection-
    based byte streams... Sockets of type SOCK_STREAM are full-duplex byte
    streams, similar to pipes.  A stream socket must be in a connected
    state before any data may be sent or received on it.  A connection to
    another socket is created with a connect(3N) call.
    ********************************************************/

   the_socket = socket (AF_INET, SOCK_STREAM, 0);
```

```

  /* set the port number */
  name.sin_family = AF_INET;
  name.sin_port = htons (port_number);
  n = INADDR_ANY;
  memcpy (&name.sin_addr, &n, sizeof (long));

  /* enables reuse of the port number -- this is a very good thing */
  temp = (int*) malloc (sizeof (int));
  *temp = 1;
  setsockopt (the_socket, SOL_SOCKET, SO_REUSEADDR, (char*)temp, sizeof(int));

  /* sets the size of the send and receive buffer -- 64 kb */
  *temp = 64; /* kilobytes */
  setsockopt (the_socket, SOL_SOCKET, SO_SNDBUF, (char *)temp, sizeof(int));
  setsockopt (the_socket, SOL_SOCKET, SO_RCVBUF, (char *)temp, sizeof(int));

  /* attempts to bind the socket -- failure returns -1 */
  res = bind (the_socket, (struct sockaddr*) (&name), sizeof (sockaddr_in));
  if (res == -1)
  {
    printf ("SHELLEY SOCKET ERROR: unable to bind socket\n");
    return (-1);
  }

  /* listen to the socket, waiting for a client to connect... if no client
     connects, then the we will listen forever, perhaps some timing mechanism
     should be implemented */

  listen (the_socket, 5);
  len = sizeof (sockaddr_in);
  ns = accept (the_socket, (struct sockaddr*) (&name), &len);

  /* if we get to this point, then a connection has been made */

  return (ns);

} /* shelley_sockets_server_listen_for_client () */




int shelley_sockets_client_connect_to_server (int port_number, char* hostname)
{
  int *temp, response, i, n, ns, player_no, the_socket, len, res;
  struct hostent *hp;
  struct sockaddr_in name;
  char buffer [50];
```

```
  the_socket = socket (AF_INET, SOCK_STREAM, 0);

  /* enable reuse of port number */
  temp = (int *) malloc (sizeof (int));
  *temp = 1;
  setsockopt (the_socket, SOL_SOCKET, SO_REUSEADDR, (char*) temp, sizeof(int));

  /* set the size of the send and receive buffer */
  *temp = 64; /* assuming kilobytes - if bytes must set to 65536 */
  setsockopt (the_socket, SOL_SOCKET, SO_SNDBUF, (char*)temp, sizeof (int));
  setsockopt (the_socket, SOL_SOCKET, SO_RCVBUF, (char*)temp, sizeof (int));

  memset (&name, 0, sizeof (struct sockaddr_in));
  name.sin_family = AF_INET;

  name.sin_port = htons (port_number);
  hp = gethostbyname (hostname);

  memcpy (&name.sin_addr, hp->h_addr_list[0], hp->h_length);
  len = sizeof (struct sockaddr_in);

  /* connect to server */
  if ((connect (the_socket, (struct sockaddr *) &name, len)) == -1)
  {
    the_socket = -1;
    printf ("SHELLEY SOCKET ERROR: client unable to connect to server\n");
  }

  return (the_socket);
} /* shelley_sockets_client_connect_to_server () */



int shelley_sockets_read (int the_socket, char *data, int length_to_read)
{
  int amount_to_read;
  int amount_read;
  int return_val = 0;;
  char *pointer;

  pointer = data;

  amount_to_read = length_to_read;

  /* loop as long as it takes to read all the data */
  while (amount_to_read > 0)
  {
```

```
    amount_read = read (the_socket, pointer, amount_to_read);
    if ((amount_read == EOF) || (amount_read == 0))
    {
      printf ("SHELLEY SOCKETS: unable to read from socket\n");
      amount_to_read = 0;
      return_val = -1;
    }
    amount_to_read -= amount_read;
    pointer += amount_read;
  }

  return (return_val);
} /* shelley_sockets_read () */



int shelley_sockets_write (int the_socket, char *data, int length_to_write)
{

  int amount_to_write = 0;
  int amount_written = 0;
  char *pointer = data;

  amount_to_write = length_to_write;

  /* loop as long as it takes to write all the data */
  while (amount_to_write > 0)
  {
    amount_written = write (the_socket, pointer, amount_to_write);
    amount_to_write -= amount_written;
    pointer += amount_written;
  }

  return (0);

} /* shelley_sockets_write () */
```

Appendix E frame_grabber_protocol.h

```
/****************************************************************************
  Andy Ritger
  Research Honors
  4-26-99

  The Frame Grabber Device Module Protocol

  Note that all constants are sent across sockets as type int8.

  ***************************************************************************/



#ifndef ___SIE_FRAME_GRABBER_PROTOCOL___
#define ___SIE_FRAME_GRABBER_PROTOCOL___



#define FRAME_GRAB_INIT_VALUE 0



/****************************************************************************
  The FRAME_GRAB_REQUEST_RESOLUTION message asks the frame grabber device
  module what resolution it is returning frames at.  This returns two int32s:
  the width and height (in pixels).
  ***************************************************************************/

#define FRAME_GRAB_REQUEST_RESOLUTION 1



/****************************************************************************
  The FRAME_GRAB_SET_RESOLUTION message is followed by a float which
  corresponds to the scale factor which will be multipled by the original
  dimensions of the image.  The FRAME_GRAB_YES message is returned upon
  success.
  ***************************************************************************/

#define FRAME_GRAB_SET_RESOLUTION 2



```

```
/*******************************************************************************
  The FRAME_GRAB_QUERY_GREYSCALE_SUPPORT message queries the device module
  if it can support grabbing frames in greyscale.  One of the messages
  FRAME_GRAB_NO or FRAME_GRAB_YES are returned in response.
  *****************************************************************************/

#define FRAME_GRAB_QUERY_GREYSCALE_SUPPORT 3



/*******************************************************************************
  The FRAME_GRAB_QUERY_RGB_SUPPORT message queries the device module if it can
  support grabbing frames in rgb color encoding.  One of the messages
  FRAME_GRAB_NO or FRAME_GRAB_YES are returned in response.
  *****************************************************************************/

#define FRAME_GRAB_QUERY_RGB_SUPPORT 4



/*******************************************************************************
  The FRAME_GRAB_NO and FRAME_GRAB_YES message are used as respones to
  requests to the device module.
  *****************************************************************************/

#define FRAME_GRAB_NO 5
#define FRAME_GRAB_YES 6



/*******************************************************************************
  The FRAME_GRAB_SELECT_INPUT_PORT message is followed by an int32 which
  specifies the port number the device should use to receive video.
  *****************************************************************************/

#define FRAME_GRAB_SELECT_INPUT_PORT 7



/*******************************************************************************
  The FRAME_GRAB_GRAB_FRAME message tells the device module to flush the video
  buffer and grab the current frame of video.  Returned is a row major stream
  of bytes, where each byte is a grey-scale pixel value.
  *****************************************************************************/

#define FRAME_GRAB_GRAB_FRAME 8
```

```
/*****************************************************************************
  The FRAME_GRAB_ENABLE_DISPLAY message tells the video grabber to show the
  grabbed frame to an X-Window.  Returns FRAME_GRAB_YES.
  ****************************************************************************/

#define FRAME_GRAB_ENABLE_DISPLAY 9



/*****************************************************************************
  The FRAME_GRAB_DISABLE_DISPLAY message tells the video grabber to not show
  the grabbed frame to an X-Window.  Returns FRAME_GRAB_YES.
  ****************************************************************************/

#define FRAME_GRAB_DISABLE_DISPLAY 10


#endif
```

Appendix F frame_grabber.c

```
/*****************************************************************************
  Andy Ritger
  Research Honors
  4-26-99

  frame_grabber.c

  Interface to sun video cameras; modified from original sample program
  included with hardware...

  *****************************************************************************/



#include <stdio.h>
#include <stdlib.h>
#include <xil/xil.h>
#include <signal.h>

#include "sie_protocol.h"
#include "shelley_sockets.h"
#include "frame_grabber_protocol.h"



/* needed by init_cmap () */

#define CMAPSIZE        256
#define TOP2   50 /* reserve the top two entries of the
  * colormap to reduce colormap flashing */



/* function prototypes */
void close_cleanly (int sig);
void rip_frame (XilImage img, unsigned char *data, int w, int h);
void init_cmap (XilLookup xil_cmap, Display * display, Window window,
 int offset);


/* global variables */
XilSystemState _xil_state;
Display *xdisplay;
```

```
/*****************************************************************************
  main ()
  *****************************************************************************/

main (int argc, char **argv)
{
  XilDevice device;
  XilImage rtvc_image, rtvc_luma, rtvc_scaled;
  XilDataType datatype;

  /* fun with Xlib */
  Window xwindow;
  XEvent event;
  int display_depth;

  int32 width, height, original_width, original_height, nbands;

  char *devname = "/dev/rtvc0";
  int max_buffers = 0;
  float scale_factor = 1.0;
  int32 port_number = 1;
  int window_shown = 0;
  int display_enabled = 0;

  unsigned char *data = NULL;

  /**************************
    open the xil library
    *************************/

  _xil_state = xil_open ();
  if (_xil_state == NULL)
  {
    fprintf (stderr, "unable to open xil library\n");
    exit (1);
  }

  /* catch "^C" so that we can close things cleanly */

  signal (SIGINT, close_cleanly);

  /* create a device so that we can set its attributes */

  if (!(device = xil_device_create (_xil_state, "SUNWrtvc")))
  {
```

```
    fprintf (stderr, "Unable to create a device object\n");
    xil_close (_xil_state);
    exit(1);
  }

  xil_device_set_value (device, "DEVICE_NAME", (void *) devname);
  xil_device_set_value (device, "MAX_BUFFERS", (void *) max_buffers);
  xil_device_set_value (device, "PORT_V", (void *) port_number);

  /* create an xil image with the above defined device values */

  if (!(rtvc_image = xil_create_from_device (_xil_state, "SUNWrtvc", device)))
  {
    fprintf (stderr, "failed to open SUNWrtvc device\n");
    xil_close (_xil_state);
    exit (1);
  }

  /* release the xil device */

  xil_device_destroy (device);

  /* get all the information about the image */

  xil_get_info (rtvc_image, &original_width, &original_height,
&nbands, &datatype);

  width = (int32) (original_width * scale_factor);
  height = (int32) (original_height * scale_factor);

  /* create a copy of the image that will have just the 1st band */

  rtvc_luma = xil_create_child (rtvc_image, 0, 0, original_width,
original_height, 0, 1);

  /* create a scaled image to put our scaled copies */

  rtvc_scaled = xil_create (_xil_state, width, height, 1, datatype);

  /* setup the Xwindow and other fun things... */

  /* xlib window creation */

  xdisplay = XOpenDisplay (NULL);

  if (!xdisplay)
  {
    fprintf (stderr, "Unable to connect to X-server\n");
```

```
    xil_close (_xil_state);
    exit (1);
  }

  display_depth = DefaultDepth (xdisplay, DefaultScreen (xdisplay));
  xwindow = XCreateSimpleWindow (xdisplay, DefaultRootWindow (xdisplay),
  0, 0, width, height, 0, 0, 0);
  if (!xwindow) {
    fprintf (stderr, "Unable to create X-window\n");
    xil_close (_xil_state);
    exit (1);
  }

  /* we'll only worry about the expose event */

  XSelectInput (xdisplay, xwindow, ExposureMask);

  /* We're operating at 8 bit display depth */

  XilLookup grayramp;
  int num_entries = 256;

  Xil_unsigned8 *graydata = (Xil_unsigned8 *) malloc (3 * num_entries);
  for (int i = 0; i < num_entries; i++)
    graydata [i * 3 + 2] = graydata [i * 3 + 1] = graydata [i * 3] = i;

  grayramp = xil_lookup_create (_xil_state, XIL_BYTE, XIL_BYTE,
 3, num_entries, 0, graydata);

  /* connect to administrator on localhost at 2048 */

  int sock = shelley_sockets_client_connect_to_server (2048, "localhost");
  if (sock == -1)
  {
    fprintf (stderr, "failed to connect to server\n");
    xil_close (_xil_state);
    exit (1);
  }

  /* hand shake with administrator */

  int8 my_val;
  shelley_sockets_read (sock, (char*) &my_val, sizeof (int8));
  if (my_val != ADMIN_QUERY_DEVICE)
  {
    fprintf (stderr, "I am confused\n");
    exit (1);
  }
```

```
  my_val = DEVICE_READY;
  shelley_sockets_write (sock, (char*) &my_val, sizeof (int8));

  /* start the request loop */

  int32 return_val;
  int8 command = FRAME_GRAB_INIT_VALUE;
  int val;
  while (command != ADMIN_DISCONNECT_DEVICE)
  {
    /* block until we get the next command */

    if ((shelley_sockets_read (sock, (char*) &command, sizeof (int8))) == -1)
      close_cleanly (0);
    else
    {
      switch (command)
      {
      case FRAME_GRAB_REQUEST_RESOLUTION:
 /* return the resolution we're using... */
 return_val = sizeof (int32) * 2;
 shelley_sockets_write (sock, (char*) &return_val, sizeof (int32));
 shelley_sockets_write (sock, (char*) &width, sizeof (int32));
 shelley_sockets_write (sock, (char*) &height, sizeof (int32));
 break;

      case FRAME_GRAB_SET_RESOLUTION:
 /* reset the scale factor */
 shelley_sockets_read (sock, (char*) &scale_factor, sizeof (float));

 /* need to error trap scale factor values */
 width = (int) (original_width * scale_factor);
 height = (int) (original_height * scale_factor);

 /* destroy the current scaled image */
 xil_destroy (rtvc_scaled);

 /* create a new scaled image, and xwindow */
 XResizeWindow (xdisplay, xwindow, width, height);
 if ((display_enabled) && (window_shown))
   rtvc_scaled = xil_create_from_window (_xil_state, xdisplay, xwindow);
 else
   rtvc_scaled = xil_create (_xil_state, width, height, 1, datatype);

 return_val = 1;
 my_val = FRAME_GRAB_YES;
 shelley_sockets_write (sock, (char*) &return_val, 4);
 shelley_sockets_write (sock, (char*) &my_val, 1);
```

```
break;

      case FRAME_GRAB_QUERY_GREYSCALE_SUPPORT:
/* yes, we do support greyscale */
my_val = FRAME_GRAB_YES;
return_val = sizeof (int8);
shelley_sockets_write (sock, (char*) &return_val, sizeof (int32));
shelley_sockets_write (sock, (char*) &val, sizeof (int8));
break;

      case FRAME_GRAB_QUERY_RGB_SUPPORT:
/* no, we do not support rgb (yet?...) */
my_val = FRAME_GRAB_NO;
return_val = sizeof (int8);
shelley_sockets_write (sock, (char*) &return_val, sizeof (int32));
shelley_sockets_write (sock, (char*) &val, sizeof (int8));
break;

      case FRAME_GRAB_SELECT_INPUT_PORT:
/* select which video port (1 or 2)... needs to be error trapped */
shelley_sockets_read (sock, (char*) &port_number, sizeof (int32));
xil_set_device_attribute (rtvc_image, "PORT_V", (void *) port_number);
break;

      case FRAME_GRAB_ENABLE_DISPLAY:
/* enable the X window */
display_enabled = 1;
return_val = 1;
my_val = FRAME_GRAB_YES;
shelley_sockets_write (sock, (char*) &return_val, 4);
shelley_sockets_write (sock, (char*) &my_val, 1);
break;

      case FRAME_GRAB_DISABLE_DISPLAY:
/* disable the X window */
display_enabled = 0;
if (window_shown)
{
  /* these two lines give focus to the xwindow, and then take the
     focus away; this is so the window manager colors are released
     when the xwindow is unmapped. */
  XSetInputFocus (xdisplay, xwindow, RevertToNone, CurrentTime);
  XSetInputFocus (xdisplay, PointerRoot, RevertToNone, CurrentTime);

  /* unmap the xwindow (hide it) */
  XUnmapWindow (xdisplay, xwindow);

```

```
      /* force any updates which need to happen */
      XFlush (xdisplay);
      window_shown = 0;
    }

    /* destroy and recreate the scaled image so that it is not connected
       to the xwindow */
    xil_destroy (rtvc_scaled);
    rtvc_scaled = xil_create (_xil_state, width, height, 1, datatype);

    return_val = 1;
    my_val = FRAME_GRAB_YES;
    shelley_sockets_write (sock, (char*) &return_val, 4);
    shelley_sockets_write (sock, (char*) &my_val, 1);
    break;

        case FRAME_GRAB_GRAB_FRAME:
    /* flush, and grab a current frame of video */
    if ((window_shown == 0) && (display_enabled))
    {
      window_shown = 1;
      XMapWindow (xdisplay, xwindow); /* make the window visible */
      do   /* wait for the window to be mapped (an Expose event) */
        XNextEvent (xdisplay, &event);
      while (event.xany.type != Expose);
      xil_destroy (rtvc_scaled);
      rtvc_scaled = xil_create_from_window (_xil_state, xdisplay, xwindow);
      init_cmap (grayramp, xdisplay, xwindow, 0);
    }

    /* flush */
    xil_set_device_attribute (rtvc_image, "FLUSH_BUFFERS", NULL);

    /* if display is connected to an xwindow, the scale draws it to
       screen; otherwise, this just makes an internal scaled copy which
       we need so that we can grab the data */

    xil_scale (rtvc_luma, rtvc_scaled, "nearest", scale_factor,
       scale_factor);
    if (data == NULL)
      data = (unsigned char *) malloc (width * height);
    rip_frame (rtvc_scaled, data, width, height);
    return_val = width*height;
    shelley_sockets_write (sock, (char*) &return_val, sizeof (int32));
    shelley_sockets_write (sock, (char *) data, (width * height));
    break;

        case ADMIN_DISCONNECT_DEVICE:
```

```
    /* we're supposed to quit, now */
    close_cleanly (0);
    break;

        } /* switch statement */

      } /* if the read succeeded */

    } /* while */
    return 0;

  } /* end main () */



/*****************************************************************************
  close_cleanly ()
  ****************************************************************************/

void close_cleanly (int sig)
{
  if (xdisplay) XCloseDisplay (xdisplay);
  xil_close (_xil_state);
  exit (0);
} /* end close_cleanly () */



/*****************************************************************************
  rip_frame ()
  ****************************************************************************/

void rip_frame (XilImage img, unsigned char *data, int w, int h)
{
  /* allocate pixel values buffer */
  float *pixel_vals = (float *) malloc (3 * sizeof (float));

  for (int y = 0; y < h; y++)
    for (int x = 0; x < w; x++)
    {
      /* get a specific pixel from the image */
      xil_get_pixel (img, x, y, pixel_vals);

      /* copy the data into our data stream to be sent to an agent */
      data [(y * w) + x] = (unsigned char) pixel_vals [0];
    }
} /* end rip_frame () */

```

```
/*****************************************************************************
  init_cmap()

  Initialize the X colormap with a 3-banded XilLookup, using the 'offset'
  argument to determine the starting pixel value in the X colormap. If the
  'offset' argument is negative, automatically calculate the starting pixel
  value in such a way as to minimize colormap flashing.

  Finally, adjust the offset of the XilLookup accordingly.
  ****************************************************************************/

void init_cmap (XilLookup xil_cmap, Display * display, Window window,
 int offset)
{
  unsigned long junk[CMAPSIZE], pixels[CMAPSIZE], mask;
  XColor cdefs[CMAPSIZE];
  Colormap rcmap;
  int cmapsize;
  int i;
  Xil_unsigned8 cmap_data[CMAPSIZE * 3];
  Xil_unsigned8 *ptr;

  rcmap = XCreateColormap(display, window,
   DefaultVisual(display, DefaultScreen(display)),
   AllocNone);

  cmapsize = xil_lookup_get_num_entries(xil_cmap);

  /* determine the offset for the colormap */
  if (offset < 0) {
    offset = 256 - cmapsize - TOP2;
    if (offset < 0)
      offset = 0; /* in case cmapsize >= 255 */
  }

  if (offset) {
    if (!XAllocColorCells(display, rcmap, 0, &mask, 0, junk, offset)) {
      fprintf (stderr, "XAlloc1 failed\n");
    }
  }

  if (!XAllocColorCells(display, rcmap, 0, &mask, 0, pixels, cmapsize)) {
    fprintf (stderr, "XAlloc2 failed\n");
  }

  /* free the unused colors in the front */
```

```
  if (offset) {
    XFreeColors (display, rcmap, junk, offset, 0);
  }

  for (i = 0; i < cmapsize; i++) {
    cdefs[i].pixel = i + offset;
  }

  xil_lookup_get_values(xil_cmap, xil_lookup_get_offset(xil_cmap),
cmapsize, cmap_data);

  ptr = cmap_data;
  for (i = 0; i < cmapsize; i++) {
    cdefs[i].flags = DoRed | DoGreen | DoBlue;

    /*
     * since 24-bit XIL images are in BGR order, colormaps are also in
     * BGR order
     */
    cdefs[i].blue = *ptr++ << 8;
    cdefs[i].green = *ptr++ << 8;
    cdefs[i].red = *ptr++ << 8;
  }
  XStoreColors(display, rcmap, cdefs, cmapsize);

  /*
   * This will cause the colormap to be installed unless the cursor is
   * moved to another window -- any other window; if this happens, then
   * colormap flashing may occur.
   */
  XSetWindowColormap(display, window, rcmap);
  XInstallColormap(display, rcmap);
  XSync(display, False);


  /* set the offset of the XilLookup */
  xil_lookup_set_offset(xil_cmap, offset);
}
```

Appendix G Frame_Grabber.H

```
/***************************************************************************
  Andy Ritger
  Research Honors
  4-26-99

  Frame_Grabber.H

  The Frame_Grabber class is a wrapper for the SIE frame grab API.

  ***************************************************************************/



#ifndef ___FRAME_GRABBER_WRAPPER___
#define ___FRAME_GRABBER_WRAPPER___



#include "sie_protocol.h"
#include "frame_grabber_protocol.h"
#include "shelley_sockets.h"


class Frame_Grabber
{
public:
  Frame_Grabber (int sock_number);
  ~Frame_Grabber () {};

  int get_width () { return width; };
  int get_height () { return height; };

  void set_scale_factor (float factor);

  void enable_display ();
  void disable_display ();

  void select_input_port (int32 port);

  void grab_frame (unsigned char *data);

 private:
  int sock;
```

```
    int32 width;
    int32 height;
    bool error;
};



#endif
```

Appendix H Frame_Grabber.C

```
/*****************************************************************************
  Andy Ritger
  Research Honors
  4-26-99

  Frame_Grabber.C

  *****************************************************************************/

#include "Frame_Grabber.H"



Frame_Grabber::Frame_Grabber (int sock_number)
{
  // initialize internal things...

  // we make the assumption that we have already connected to the admin
  sock = sock_number;
  error = false;

  // what we're doing
  int8 val = AGENT_SEND_DEVICE;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));

  // to which device
  int32 val32 = 1;
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // how long the message is
  val32 = sizeof (int8);
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // the message (it's about time)
  val = FRAME_GRAB_REQUEST_RESOLUTION;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));

  // the next thing coming back is the resolution
  shelley_sockets_read (sock, (char*) &width, sizeof (int32));
  shelley_sockets_read (sock, (char*) &height, sizeof (int32));

} // constructor

```

```
void Frame_Grabber::set_scale_factor (float factor)
{
  if (sock == -1) return;

  // what we're doing
  int8 val = AGENT_SEND_DEVICE;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));

  // to which device
  int32 val32 = 1;
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // how long the message is
  val32 = sizeof (int8) + sizeof (float);
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // the message (finally)
  val = FRAME_GRAB_SET_RESOLUTION;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));
  shelley_sockets_write (sock, (char*) &factor, sizeof (float));

  // frame grab returns a yes or no...
  shelley_sockets_read (sock, (char*) &val, sizeof (int8));

  /* now get the new width and height */

  // what we're doing
  val = AGENT_SEND_DEVICE;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));

  // to which device
  val32 = 1;
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // how long the message is
  val32 = sizeof (int8);
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // the message (it's about time)
  val = FRAME_GRAB_REQUEST_RESOLUTION;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));

  // the next thing coming back is the resolution
  shelley_sockets_read (sock, (char*) &width, sizeof (int32));
  shelley_sockets_read (sock, (char*) &height, sizeof (int32));

```

```
} // set_scale_factor ()



void Frame_Grabber::enable_display ()
{
  if (sock == -1) return;

  // what we're doing
  int8 val = AGENT_SEND_DEVICE;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));

  // to which device
  int32 val32 = 1;
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // how long the message is
  val = sizeof (int8);
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // the message (it's about time)
  val = FRAME_GRAB_ENABLE_DISPLAY;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));

  // frame grab returns a yes or no...
  shelley_sockets_read (sock, (char*) &val, sizeof (int8));

} // enable_display ()




void Frame_Grabber::disable_display ()
{
  if (sock == -1) return;

  // what we're doing
  int8 val = AGENT_SEND_DEVICE;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));

  // to which device
  int32 val32 = 1;
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // how long the message is
  val32 = sizeof (int8);
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

```

```
  // the message (it's about time)
  val = FRAME_GRAB_DISABLE_DISPLAY;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));

  // frame grab returns a yes or no...
  shelley_sockets_read (sock, (char*) &val, sizeof (int8));

} // disable_display ()




void Frame_Grabber::select_input_port (int32 port)
{
  if (sock == -1) return;

  // what we're doing
  int8 val = AGENT_SEND_DEVICE;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));

  // to which device
  int32 val32 = 1;
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // how long the message is
  val32 = sizeof (int8) + sizeof (int32);
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // the message (it's about time)
  val = FRAME_GRAB_SELECT_INPUT_PORT;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));
  shelley_sockets_write (sock, (char*) &port, sizeof (int32));

  // frame grab returns a yes or no...
  shelley_sockets_read (sock, (char*) &val, sizeof (int8));

} // select_input_port ()




void Frame_Grabber::grab_frame (unsigned char *data)
{
  if (sock == -1) return;
  if (data == NULL) return;

  // what we're doing
  int8 val = AGENT_SEND_DEVICE;
```

```
   shelley_sockets_write (sock, (char*) &val, sizeof (int8));

   // to which device
   int32 val32 = 1;
   shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

   // how long the message is
   val32 = sizeof (int8);
   shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

   val = FRAME_GRAB_GRAB_FRAME;
   shelley_sockets_write (sock, (char*) &val, sizeof (int8));

   /* read the BIG 1-d array of chars... */
   shelley_sockets_read (sock, (char *) data, (width * height));

} // grab_frame ()
```

Appendix I neural_net_protocol.h

```
/***************************************************************************
  Andy Ritger
  Research Honors
  4-26-99

  The Artificial Neural Network Device Module Protocol

  Note that all constants are sent across sockets as type int8.

  ***************************************************************************/



#ifndef ___SIE_NEURAL_NETWORK_PROTOCOL___
#define ___SIE_NEURAL_NETWORK_PROTOCOL___



#define NEURAL_NET_INIT_VALUE 0



/***************************************************************************
  NEURAL_NET_CREATE is followed by three 32-bit integers: the number of input,
  hidden and output nodes for the new network.
 ***************************************************************************/

#define NEURAL_NET_CREATE 1



/***************************************************************************
  NEURAL_NET_FEED_FORWARD computes an answer(s) for the network by applying
  the feed forward algorithm to the input layer and arriving at values at
  the hidden layer, and similarly using the hidden layer to arrive at values
  at the output layer.
  ***************************************************************************/

#define NEURAL_NET_FEED_FORWARD 2



/***************************************************************************
```

```
  NEURAL_NET_LOAD_INPUT_VECTOR is followed by a stream of data which
  corresponds to the double-precision values of the input layer.  The
  data stream is row-major (it can be indexed with: ((y * maxx) + x)).
 ****************************************************************************/

#define NEURAL_NET_LOAD_INPUT_VECTOR 3



/****************************************************************************
  NEURAL_NET_LOAD_TARGET_VECTOR is followed by a 32-bit index value indicating
  which output node we're talking about, which is then followed by the double
  value which we want to assign to that target node.
 ****************************************************************************/

#define NEURAL_NET_LOAD_TARGET_VALUE 4



/****************************************************************************
  NEURAL_NET_TRAIN trains the network.  NEURAL_NET_YES is returned when
  training is complete.
 ****************************************************************************/

#define NEURAL_NET_TRAIN 5



/****************************************************************************
  NEURAL_NET_GET_OUTPUT_VALUE is followed by a 32-bit integer which
  indicates which output node is in question.  A double precision value
  is returned indicating the value of that node.
 ****************************************************************************/

#define NEURAL_NET_GET_OUTPUT_VALUE 6



/****************************************************************************
  Yes and No responses from the Neural Network
 ****************************************************************************/

#define NEURAL_NET_YES 7
#define NEURAL_NET_NO  8



#endif
```

Appendix J neural_net.c

```
/***************************************************************************
  Andy Ritger
  Research Honors
  4-26-99

  neural_net.C

  This is the artificial neural network device module.  We initiate things,
  and then listen for commands.  This interfaces with the bpnn class.

  ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>



#include "sie_protocol.h"
#include "neural_net_protocol.h"
#include "shelley_sockets.h"
#include "bpnn.H"



int main (int argc, char **argv)
{
  bpnn *neural_net;
  int32 input, hidden, output;
  int32 val32;
  int8  val8;

  double *data;
  double value;

  double output_error, hidden_error;

  // connect to administrator on localhost at 2048
  int sock = shelley_sockets_client_connect_to_server (2048, "localhost");
  if (sock == -1)
  {
    cout << "NEURAL NET: failed to connect to administrator." << endl;
```

```
    exit (1);
  }

  // hand shake with administrator
  shelley_sockets_read (sock, (char*) &val8, sizeof (int8));
  if (val8 != ADMIN_QUERY_DEVICE)
  {
    cout << "NEURAL NET: I am confused." << endl;
    exit (1);
  }
  val8 = DEVICE_READY;
  shelley_sockets_write (sock, (char*) &val8, sizeof (int8));

  // now we can expect to receive neural net protocol commands
  int8 command = NEURAL_NET_INIT_VALUE;

  // loop in here until we're told to exit
  while (command != ADMIN_DISCONNECT_DEVICE)
  {
    // block until we get the next command
    if ((shelley_sockets_read (sock, (char*) &command, sizeof (int8))) == -1)
      exit (1);
    else
    {
      // perform some action based on what the request is
      switch (command)
      {
      case NEURAL_NET_CREATE:
// create a new network, given the dimensions

shelley_sockets_read (sock, (char*) &input, sizeof (int32));
shelley_sockets_read (sock, (char*) &hidden, sizeof (int32));
shelley_sockets_read (sock, (char*) &output, sizeof (int32));

neural_net = new bpnn (input, hidden, output);
neural_net->initialize (false, true, 0.0); // yes, it's hardcoded...

// confirm that we received the data and that the network is created

val32 = 1;
val8  = NEURAL_NET_YES;
shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
break;

      case NEURAL_NET_FEED_FORWARD:
// feed what is in the input layer through the network
neural_net->feedforward ();
```

```
// confirm that we did apply feedforward
val32 = 1;
val8  = NEURAL_NET_YES;
shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
break;

      case NEURAL_NET_LOAD_INPUT_VECTOR:
// load the input vector
data = new double [input];
shelley_sockets_read (sock, (char*) data, sizeof (double) * input);

for (int i = 0; i < input; i++)
  neural_net->load_input_value (i, data [i]);

delete (data);

// confirm that we did load the input vector
val32 = 1;
val8  = NEURAL_NET_YES;
shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
break;

      case NEURAL_NET_LOAD_TARGET_VALUE:
// set the target values
shelley_sockets_read (sock, (char*) &val32, sizeof (int32)); // index
shelley_sockets_read (sock, (char*) &value, sizeof (double)); // value

neural_net->load_target_value (val32, value);

// confirm that we did load the target value
val32 = 1;
val8  = NEURAL_NET_YES;
shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
break;

      case NEURAL_NET_TRAIN:
// train the network

neural_net->train (0.3, 0.3, &output_error, &hidden_error);

// confirm that we did train the network
val32 = 1; // our response will be 1 byte
val8  = NEURAL_NET_YES;
shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
```

```
    shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
    break;

         case NEURAL_NET_GET_OUTPUT_VALUE:
    // return the value of an output node
    shelley_sockets_read (sock, (char*) &val32, sizeof (int32)); // index
    value = neural_net->get_output_value (val32);

    // send back the value
    val32 = sizeof (double);
    shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
    shelley_sockets_write (sock, (char*) &value, sizeof (double));

    break;

         case ADMIN_DISCONNECT_DEVICE:
    // we should exit now
    exit (0);
    break;

         } // switch

       } // if

     } // while

   } // main ()
```

Appendix K Neural_Net.H

```
/******************************************************************************
  Andy Ritger
  Research Honors
  4-26-99

  Neural_Net.H

  The Neural_Net class is a wrapper for the SIE neural net API.

  ******************************************************************************/



#ifndef ___NEURAL_NET_WRAPPER___
#define ___NEURAL_NET_WRAPPER___



#include "sie_protocol.h"
#include "neural_net_protocol.h"
#include "shelley_sockets.h"



class Neural_Net
{
public:
  Neural_Net (int sock_number);
  ~Neural_Net () {};

  void feedforward ();
  void load_input_vector (double* data);
  void load_target_value (int32 index, double value);
  void train ();
  double get_output_value (int32 index);


private:
  int sock;
  bool error;
};
```

```
#endif
```

Appendix L Neural_Net.C

```
/******************************************************************************
  Andy Ritger
  Research Honors
  4-26-99

  Neural_Net.C

  ******************************************************************************/

#include "Neural_Net.H"



#define _SIZE_ 9612 // this is only a temporary fix - the number of inputs



Neural_Net::Neural_Net (int sock_number)
{
  int8 val8;
  int32 val32;

  // initialize internal things

  // we make the assumption that we have already connected to the admin
  sock = sock_number;
  error = false;

  // what we're doing
  val8 = AGENT_SEND_DEVICE;
  shelley_sockets_write (sock, (char*) &val8, sizeof (int8));

  // to which device
  val32 = 2;
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // how long the message is
  val32 = (sizeof (int8)) + (sizeof (int32) * 3);
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // the message (it's about time)
  val8 = NEURAL_NET_CREATE;
  shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
```

```
  val32 = _SIZE_; // input layer
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  val32 = 4; // hidden layer
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  val32 = 4; // output layer
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // neural net returns a yes or no...
  shelley_sockets_read (sock, (char*) &val8, sizeof (int8));

} // constructor




void Neural_Net::feedforward ()
{
  int8 val8;
  int32 val32;

  // what we're doing
  val8 = AGENT_SEND_DEVICE;
  shelley_sockets_write (sock, (char*) &val8, sizeof (int8));

  // to which device
  val32 = 2;
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // how long the message is
  val32 = sizeof (int8);
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // the message (it's about time)
  val8 = NEURAL_NET_FEED_FORWARD;
  shelley_sockets_write (sock, (char*) &val8, sizeof (int8));

  // neural net returns a yes or no...
  shelley_sockets_read (sock, (char*) &val8, sizeof (int8));

} // feedforward ()


void Neural_Net::load_input_vector (double* data)
{
  int8 val8;
```

```
  int32 val32;

  // what we're doing
  val8 = AGENT_SEND_DEVICE;
  shelley_sockets_write (sock, (char*) &val8, sizeof (int8));

  // to which device
  val32 = 2;
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // how long the message is
  val32 = (sizeof (double) * _SIZE_) + (sizeof (int8));
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // the message (it's about time)
  val8 = NEURAL_NET_LOAD_INPUT_VECTOR;
  shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
  shelley_sockets_write (sock, (char*) data, sizeof sizeof(double) * _SIZE_);

  // neural net returns a yes or no...
  shelley_sockets_read (sock, (char*) &val8, sizeof (int8));

} // load_input_vector ()



void Neural_Net::load_target_value (int32 index, double value)
{
  int8 val8;
  int32 val32;

  // what we're doing
  val8 = AGENT_SEND_DEVICE;
  shelley_sockets_write (sock, (char*) &val8, sizeof (int8));

  // to which device
  val32 = 2;
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // how long the message is
  val32 = (sizeof (double)) + (sizeof (int32)) + (sizeof (int8));
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // the message (it's about time)
  val8 = NEURAL_NET_LOAD_TARGET_VALUE;

  shelley_sockets_write (sock, (char*) &val8, sizeof (int8)); // command
  shelley_sockets_write (sock, (char*) &index, sizeof (int32)); // index
```

```
  shelley_sockets_write (sock, (char*) &value,  sizeof (double)); // value

  // neural net returns a yes or no...
  shelley_sockets_read (sock, (char*) &val8, sizeof (int8));

} // load_target_value ()




void Neural_Net::train ()
{
  int8 val8;
  int32 val32;

  // what we're doing
  val8 = AGENT_SEND_DEVICE;
  shelley_sockets_write (sock, (char*) &val8, sizeof (int8));

  // to which device
  val32 = 2;
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // how long the message is
  val32 = sizeof (int8);
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // the message (it's about time)
  val8 = NEURAL_NET_TRAIN;
  shelley_sockets_write (sock, (char*) &val8, sizeof (int8));

  // neural net returns a yes or no...
  shelley_sockets_read (sock, (char*) &val8, sizeof (int8));

} // train ()



double Neural_Net::get_output_value (int32 index)
{
  int8 val8;
  int32 val32;

  // what we're doing
  val8 = AGENT_SEND_DEVICE;
  shelley_sockets_write (sock, (char*) &val8, sizeof (int8));

  // to which device
```

```
    val32 = 2;
    shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

    // how long the message is
    val32 = sizeof (int8) + sizeof (int32);
    shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

    // the message (it's about time)
    val8 = NEURAL_NET_GET_OUTPUT_VALUE;
    shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
    shelley_sockets_write (sock, (char*) &index, sizeof (int32)); // index

    double value;

    // neural net returns double
    shelley_sockets_read (sock, (char*) &value, sizeof (double));

    return (value);

  } // get_output_value ()
```

Appendix M agent.C

```
/*****************************************************************************
  Andy Ritger
  Research Honors
  4-26-99

  agent.C

  This is an example agent which initially presents the user with options to:

  [1] identify user
  [2] capture frames to pgm
  [3] train
  [4] quit

  Identifying the user (1) grabs a frame of video, feeds it into the input of
  the neural network, and simply reports the output values.

  Capturing frames of video to pgm (2) grabs X number of video frames, and
  saves them as pgm images for future use in training.

  Training (3) uses the pgm images and trains the network to recognize the
  faces of the people in the pgms.



  More important than the face recognition functionality that this agent
  provides, this program demonstrates a simple example of using SIE and
  how an agent should interact with the administrator and send requests to
  devices.

  *****************************************************************************/



#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>

#include "sie_protocol.h"
#include "shelley_sockets.h"
#include "Frame_Grabber.H"
#include "Neural_Net.H"
```

```
#include "pgmImage.H"



#define _SIZE_ 6912 // this is a temporary fix -- the size of the network input



/****************************************************************************
  Given a byte stream of data, writes a pgm image to disk.
  ****************************************************************************/

void write_pgm_file (char* filename, unsigned char *data,
      int width, int height)
{
  // streams are better than file handlers

  ofstream *foobar;
  foobar = new ofstream (filename);

  *foobar << "P2" << endl;
  *foobar << width << " " << height << endl;
  *foobar << "255" << endl;

  for (int y = 0; y < height; y++)
  {
    for (int x = 0; x < width; x++)
      *foobar << (int) data [(y * width) + x] << " ";
    *foobar << endl;
  }
  foobar->close();
} // write_pgm_file ()



/****************************************************************************
  Requests a frame of video, passes it to the input of the neural network,
  and prints the resulting outputs of the network.
  ****************************************************************************/

void identify (Frame_Grabber *grabber, Neural_Net *network)
{
  // get the current dimensions of the frame
  int size = grabber->get_width () * grabber->get_height ();

  // allocate memory for the data
  unsigned char *data = new unsigned char [size];
  double *inputs = new double [size];
```

```

  // allow the frame to be shown to the screen, and
  grabber->enable_display ();
  grabber->grab_frame (data);

  // convert from chars to doubles 0.0 >= x > 1.0
  for (int i = 0; i < size; i++)
    inputs [i] = ((double) data [i]) / ((double) 256.0);

  // feed the converted frame data into the network's inputs
  network->load_input_vector (inputs);
  network->feedforward ();

  // print the results
  for (int i = 0; i < 4; i++)
    cout << "  [" << i << "] = " << network->get_output_value (i);
  cout << endl;

  // disable the video display
  grabber->disable_display ();

  // free the memory that we allocated
  delete (data);
  delete (inputs);

} // identify ()



/*****************************************************************************
  Reads image.list and loads the specified images, training the network
  to recognize each.  For more details on the network is trained, see the
  bpnn.H documentation, as well as documentation relating to the GNNV
  project (www.iwu.edu/~shelley/gnnv).
  *****************************************************************************/

void train (Neural_Net *network)
{
  // load the images
  pgmImageList *list = new pgmImageList ("image.list");

  cout << "number of epochs: ";
  int max_epochs = 10;
  cin >> max_epochs;

  int epoch;
  double output_error, hidden_error, error_sumation;
  int numcorrect = 0;
```

```
    double sum_error = 0.0;
    double value;
    double *inputs = new double [_SIZE_];
    pgmImage *img;
    double answers [4];
    int index;

    // for each epoch, we examine all the images in the image list
    for (epoch = 0; epoch < max_epochs; epoch++)
    {
      numcorrect = 0;
      for (int i = 0; i < list->numberOfImages (); i++)
      {
        img = list->getImage (i);

        // load the input vector
        index = 0;
        for (int j = 0; j < img->rows; j++)
   for (int k = 0; k < img->cols; k++)
   {
     inputs [index]  = ((double) (img->getPixel (j, k))) / 256.0;
     index++;
   }

        // feed the input data to the network
        network->load_input_vector (inputs);

        /* load the target vector*/

        // start all targets low
        network->load_target_value (0, 0.1);
        network->load_target_value (1, 0.1);
        network->load_target_value (2, 0.1);
        network->load_target_value (3, 0.1);

        /*
   For this test, we use the simple convention where the name of all
   image for person 1 begin with the number 1.  For example, a pgm
   filename may be: 1.4.pgm, which means that it is picture number 4 for
   person 1.  This way, we can just look at the first character of the
   name when we want to load the target vector.
   */

         if (img->basefilename [0] == '0')
   network->load_target_value (0, 0.9);
         else if (img->basefilename [0] == '1')
   network->load_target_value (1, 0.9);
         else if (img->basefilename [0] == '2')
```

```
network->load_target_value (2, 0.9);
    else if (img->basefilename [0] == '3')
network->load_target_value (3, 0.9);

    // train...
    network->train;

    // count for ourselves
    answers [0] = network->get_output_value (0);
    answers [1] = network->get_output_value (1);
    answers [2] = network->get_output_value (2);
    answers [3] = network->get_output_value (3);

    if ((img->basefilename [0] == '0') &&
  (answers [0] > 0.5))
numcorrect++;

    else if ((img->basefilename [0] == '1') &&
      (answers [1] > 0.5))
numcorrect++;

    else if ((img->basefilename [0] == '2') &&
      (answers [2] < 0.5))
numcorrect++;

    else if ((img->basefilename [0] == '3') &&
      (answers [3] < 0.5))
numcorrect++;

   } // each image

   cout << "epoch: " << epoch << " " << numcorrect << " correct" << endl;

  } // each epoch

  // free all the memory we allocated
  delete (inputs);
  delete (list);

} // train ()



/****************************************************************************
  Here we grab frames of video, and send the data to the write_pgm_file ()
  function to save it in pgm file format.
  ****************************************************************************/

```

```
void capture_frames (Frame_Grabber *grabber)
{
  // get the user's number (0-3)
  cout << "please enter your number [0-3]: ";
  int name;
  cin >> name;

  // get the number of frames to grab (which is how many pgms will be saved)
  cout << "please enter the number of frames to grab: ";
  int frames;
  cin >> frames;

  char filename [50];

  // this is where we put the frames
  unsigned char *data = (unsigned char *) malloc (grabber->get_width () *
   grabber->get_height ());

  // enable the video display of the grabbed frame
  grabber->enable_display ();

  // for each frame we want to grab
  for (int i = 0; i < frames; i++)
  {
    // get the data
    grabber->grab_frame (data);

    // make the filename
    sprintf (filename, "images/%d.%d.pgm", name, i);

    // send the data off to be written to disk
    write_pgm_file (filename, data, grabber->get_width (),
     grabber->get_height ());

    // pause for 1 second
    sleep (1);
  }

  // disable the display
  grabber->disable_display ();

  // free the memory that we allocated
  free (data);

} // capture frames
```

```
/*****************************************************************************
  main ()
  *****************************************************************************/

int main (int argc, char **argv)
{
  // connect to the administrator; sock is the socket identifier
  int sock = shelley_sockets_client_connect_to_server (2048, "localhost");

  // tell the administrator that we're an agent
  int8 val = AGENT_CONNECT;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));

  // this is the administrator telling us that he is an administrator
  shelley_sockets_read (sock, (char*) &val, sizeof (int8));

  // request devices...

  val = AGENT_DEVICE_REQUEST;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));
  int32 val32 = 1; // request frame grabber
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // eventually we'll need to listen for a response

  val = AGENT_DEVICE_REQUEST;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));
  val32 = 2; // request neural net
  shelley_sockets_write (sock, (char*) &val32, sizeof (int32));

  // eventually we'll need to listen for a response

  // tell the administrator we're done requesting devices
  val = AGENT_DEVICE_REQUEST_DONE;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));

  // talk to the frame grabber
  Frame_Grabber *grabber = new Frame_Grabber (sock);
  int width, height;
  width = grabber->get_width ();
  height = grabber->get_height ();
  cout << "Initial frame dimensions are "
       << width << " x " << height
       << endl;
  grabber->set_scale_factor (0.15);
  width = grabber->get_width ();
  height = grabber->get_height ();
  cout << "Frames resized to "
```

```
        << width << " x " << height
        << endl;

  // talk to the neural network
  Neural_Net *network = new Neural_Net (sock);

  // present the user with our menu
  int choice = 0;

  while (choice != 4)
  {
    printf ("[1] identify user\n");
    printf ("[2] capture frames to pgm\n");
    printf ("[3] train\n");
    printf ("[4] quit\n");

    cout << "choice: ";
    cin >> choice;

    if (choice == 1)
      identify (grabber, network);

    else if (choice == 2)
      capture_frames (grabber);

    else if (choice == 3)
      train (network);
  }

  // disconnect from the administrator
  val = AGENT_DISCONNECT;
  shelley_sockets_write (sock, (char*) &val, sizeof (int8));

} // main ()
```

Appendix N Makefile

```
# Andy Ritger
# Research Honors
# 4-26-99

# Makefile for demonstation of SIE

CC =                g++
LIBS =                -lsocket -lnsl
XILHOME =               /opt/SUNWits/Graphics-sw/xil
CFLAGS =              -I$(OPENWINHOME)/include
XILLIBS =              -L$(XILHOME)/lib -L$(OPENWINHOME)/lib\
                       -lxil -lX11 -ldl -ldga -lm -lthread\
                       -R $(XILHOME)/lib:/usr/openwin/lib\
                      -lsocket -lnsl

all:                  administrator agent frame_grabber neural_net

clean:
                      rm -f core *.o *~

administrator:        administrator.c sie_protocol.h
                      $(CC) $(LIBS) administrator.c -o administrator

agent:                agent.C shelley_sockets.o Frame_Grabber.o \
                      Neural_Net.o sie_protocol.h pgmImage.o
                      $(CC) $(LIBS) shelley_sockets.o agent.C  \
                      Frame_Grabber.o Neural_Net.o pgmImage.o -o $@

frame_grabber:         frame_grabber.o shelley_sockets.o
                       $(CC) $(XILLIBS) frame_grabber.o  \
                       shelley_sockets.o -o $@

neural_net:           neural_net.C neural_net_protocol.h bpnn.o \
                      shelley_sockets.o
                      $(CC) $(LIBS) neural_net.C bpnn.o  \
                      shelley_sockets.o -o $@

shelley_sockets.o:     shelley_sockets.c shelley_sockets.h
                       $(CC) -c $<

frame_grabber.o:       frame_grabber.c frame_grabber_protocol.h sie_protocol.h
                       $(CC) $(CFLAGS) -c $<
```

```
Frame_Grabber.o:        Frame_Grabber.C Frame_Grabber.H
                        $(CC) $(CFLAGS) -c $<

Neural_Net.o:            Neural_Net.C Neural_Net.H
                         $(CC) $(CFLAGS) -c $<

bpnn.o:                  bpnn.C bpnn.H
                        $(CC) $< -c

pgmImage.o:             pgmImage.C pgmImage.H
                        $(CC) $< -c
```

References

[1] Stuart Russell and Peter Norvig, *Artificial Intelligence, A Modern Approach* (Prentice-Hall 1995).

[2] *Illinois Wesleyan Intelligence Network on Knowledge, member Cognitive Science Consortium.* http://www.iwu/edu/~lshapiro/wink.html

[3] Abraham Silberschatz and Peter B. Galvin, *Operating System Concepts* (Addison-Wesley 1995).

[4] Graham Glass, *Unix for Programmers and Users* (Prentice Hall 1993).

[5] W. Richard Stevens, *TCP/IP Illustrated, Vol. 1: The Protocols* (Addison-Wesley 1994).

[6] W. Richard Stevens, *Unix Network Programming: Interprocess Communications Vol. 2*, Second Edition (Prentice Hall 1999).

[7] Douglas Gage, Network Protocols for Mobile Robot Systems, SPIE Mobile Robots XII, Pittsburgh, PA (October 1997).

[8] Bjarne Stroustrup, AT&T Labs, *The C++ Programming Language*, 3rd Edition (Addison-Wesley 1997).

[9] Joseph L. Jones and Anita M. Flynn, *Mobile Robots, Inspiration to Implementation* (A K Peters 1993).

[10] Rodney A. Brooks, *Elephants Don't Play Chess*, Robotics and Autonomous Systems 6 (MIT Press 1990).

[11] Tom M. Mitchell, *Machine Learning* (McGraw-Hill 1997).

Acknowledgements

This project was made possible in part by the IWU Mellon Center, Instructional Technology Grant in support of IWINK/The SHELLEY RESEARCH GROUP. I would like to extend a special thanks to The SHELLEY RESEARCH GROUP, the faculty members of my research hearing committee, and especially Dr. Lionel Shapiro for all their advice and support during the course of this research.